Uspořádaný slovníkový problém

Jedná se o rozšíření základního slovníkového problému. Je dáno totálně uspořádané univerzum U (tj. pro každé dva různé prvky $u, v \in U$ platí buď $u < v$ nebo $v < u$). Cílem je reprezentovat množinu $S \subseteq U$ a navrhnout algoritmy pro tyto operace:
MEMBER, **INSERT**, **DELETE**
MIN – nalezne nejmenší prvek v S,
MAX – nalezne největší prvek v S,
SPLIT (x) – zkonstruuje reprezentace množin $S_1 = \{s \in S \mid s < x\}$ a $S_2 = \{s \in S \mid s > x\}$ a oznámí, zda $x \in S$,
JOIN – používají se dvě verze této operace:
JOIN2 (S_1, S_2) – jsou dány reprezentace množin S_1 a S_2, které splňují $\max S_1 < \min S_2$, vytvoří se reprezentace množiny $S = S_1 \cup S_2$,
JOIN3 (S_1, x, S_2) – jsou dány reprezentace množin S_1 a S_2 a prvek $x \in U$ tak, že je splněno $\max S_1 < x < \min S_2$, vytvoří se reprezentace množiny $S = S_1 \cup \{x\} \cup S_2$.

Je vidět, že operace **JOIN2** a **JOIN3** lze pomocí operací **INSERT** a **DELETE** převést jednu na druhou. Proto často budeme popisovat pro danou strukturu jen jednu z nich. Občas se také používá operace
ord (k) – předpokádáme, že $k \leq |S|$, a operace nalezne k-tý nejmenší prvek v S.
Zřejmě operace **MIN** a **MAX** jsou speciálním případem operace **ord**(k), přesně **MIN** je operace **ord** (1) a **MAX** je operace **ord** $(|S|)$.

(a, b)-stromy

Důležitou datovou strukturou vhodnou pro řešení uspořádaného slovníkového problému jsou (a, b)-stromy. Tuto datovou strukturu lze použít pro interní i pro externí paměť. Je to struktura založená na stromech. Nejobecnější grafová definice (a, b)-stromu je:
Nechť $1 \leq a < b$ jsou kladná přirozená čísla. Pak kořenový strom (T, t) se nazývá (a, b)-strom, když

(1) když v je vnitřní vrchol stromu T různý od kořene t, pak má alespoň a a nejvýše b synů;
(2) všechny cesty z kořene do libovolného listu mají stejnou délku.

Tato definice je příliš obecná a pro datové struktury se nehodí. Proto používáme její speciální případ. Datová struktura (a, b)-strom je definována jen na těchto stromech: Nechť a a b jsou přirozená čísla taková, že $2 \leq a$ a $2a - 1 \leq b$. Pak kořenový strom (T, t) nazveme (a, b)-strom, když platí

(1) každý vnitřní vrchol v stromu T různý od kořene t má alespoň a a nejvýše b synů;
(2) kořen je buď list nebo má alespoň dva syny a nejvýše b synů;
(3) všechny cesty z kořene do libovolného listu mají stejnou délku.

Výhody našich (a, b)-stromů:
Když má (a, b)-strom výšku $h > 0$ (tj. délka každé cesty z kořene do libovolného listu je h), pak strom má alespoň $2a^{h-1}$ listů a nejvýše b^h listů.

Tvrzení. *Mějme přirozená čísla a a b taková, že $a \geq 2$ a $b \geq 2a - 1$. Pak pro každé kladné přirozené číslo n existuje (a, b)-strom, který má přesně n listů. Když (a, b)-strom má přesně n listů, pak výška stromu je nejvýše $1 + \log_a \left(\frac{n}{2}\right)$ a je alespoň $\log_b n$. Tedy výška stromu je $O(\log n)$.*

Mějme kořenový strom (T, t) takový, že pro každý vnitřní vrchol v platí:
když v má $\rho(v)$ synů, pak jsou očíslovány od 1 do $\rho(v)$. Řekneme, že vrchol v je v hloubce

h, když cesta z kořene t do v má délku h. Množina všech vrcholů v hloubce h se nazývá h-tá hladina. Lexikografické uspořádání na h-té hladině je definováno rekurzivně:

$v \leq w$, právě když buď $\mathrm{otec}(v) < \mathrm{otec}(w)$ nebo $\mathrm{otec}(v) = \mathrm{otec}(w)$ a když v je i-tý syn $\mathrm{otec}(v)$ a w je j-tý syn $\mathrm{otec}(v)$, pak $i \leq j$.

Předpokládáme, že v (a,b)-stromu synové každého vnitřního vrcholu jsou uspořádány. Listy tvoří hladinu h, kde h je hloubka (a,b)-stromu, a je na nich definováno lexikografické uspořádání.

Mějme lineárně uspořádané univerzum U a množinu $S \subseteq U$. Pak (a,b)-strom (T,t) reprezentuje množinu S, když má přesně $|S|$ listů a je dán izomorfismus mezi lexikografickým uspořádáním listů stromu T a uspořádanou množinou S (tj. bijekce $\mathrm{key} : \mathrm{list}(T) \to S$, která pro $s,t \in S$ splňuje $s \leq t$ v U, právě když $\mathrm{key}^{-1}(s) \leq \mathrm{key}^{-1}(t)$ v lexikografickém uspořádání na množině listů stromu T).

Struktura vnitřních vrcholů (a,b)-stromu (T,t) reprezentujícího množinu $S \subseteq U$:
$\rho(v)$ – počet synů vrcholu v,
$S_v(1..\rho(v))$ – pole ukazatelů na syny vrcholu v takové, že $S_v(i)$ je i-tý syn vrcholu v pro $i = 1, 2 \ldots, \rho(v)$,
$H_v(1..\rho(v) - 1)$ – pole prvků z U takové, že $H_v(i)$ je největší prvek z S reprezentovaný v podstromu i-tého syna vrcholu v (alternativa: $H_v(i)$ je prvek z U takový, že největší prvek reprezentovaný v podstromu i-tého syna vrcholu v je menší nebo roven $H_v(i)$ a to je menší než nejmenší prvek reprezentovaný v podstromu $(i+1)$-ního syna vrcholu v).

Struktura listů:
listu v je přiřazen prvek $\mathrm{key}(v) \in S$.

Někdy je ve struktuře každého vrcholu v (a,b)-stromu různého od kořene ještě ukazatel $\mathrm{otec}(v)$ na otce vrcholu v.

Když $H_v(i)$ jsou prvky z reprezentované množiny, pak pro každý prvek $s \in S$ kromě největšího existuje právě jeden vnitřní vrchol v (a,b)-stromu a jedno i, že $H_v(i) = s$, a největší prvek v S není prvek H_v pro žádný vrchol v. Tento fakt se používá při implementaci, kde se vynechávají listy. Prvky z S jsou reprezentovány v polích H_v vnitřních vrcholů stromu a největší prvek je uložen zvlášť nebo je k množině S přidán formální největší prvek (a ten je pak "uložen" zvlášť). Je to prostorově efektivnější reprezentace množiny S, ale je technicky nepřehledná. Proto při práci s (a,b)-stromy používám verzi s listy.

Nyní uvedeme algoritmy pro (a,b)-stromy.

Algoritmy.
Pomocný algoritmus

Vyhledej(x)
t :=kořen stromu T, $w := NIL$
while t není list **do**
  $i := 1$
  while $H_t(i) < x$ a $i < \rho(t)$ **do** $i := i + 1$ **enddo**
  if $H_t(i) = x$ **then** $w := t$ **endif**
$t := S_t(i)$ **enddo** **Výstup**: t a w.

MEMBER(x)
Vyhledej(x)
if key $(t) = x$ **then Výstup**: $x \in S$ **else Výstup**: $x \notin S$ **endif**

INSERT(x)
Vyhledej(x)
if key $(t) \neq x$ **then**
vytvoř nový list t', key $(t') := x$, $u :=$ otec (t)
if key $(t) < x$ **then**
(komentář: $x > \max S$)
$S_u(\rho(u)+1) := t'$, $H_u(\rho(u)) :=$ key (t), $\rho(u) := \rho(u)+1$
else
najdi i, že $S_u(i) = t$
$S_u(\rho(u)+1) := S(\rho(u))$, $j := \rho(u) - 1$
while $j \geq i$ **do**
$S_u(j+1) := S_u(j)$, $H_u(j+1) := H_u(j)$, $j := j-1$
enddo
$S_u(i) := t'$, $H_u(i) := x$, $\rho(u) := \rho(u)+1$
endif
$t := u$
while $\rho(t) > b$ **do Štěpení**(t) **enddo**
endif

Štěpení(t)
if t je kořen stromu **then**
vytvoř nový kořen u s jediným synem t
endif
$u :=$ otec (t), najdi i, že $S_u(i) = t$
vytvoř nový vnitřní vrchol t', $j := 1$
while $j < \lfloor \frac{b+1}{2} \rfloor$ **do**
$S_{t'}(j) := S_t\left(j + \lceil \frac{b+1}{2} \rceil\right)$, $H_{t'}(j) := H_t\left(j + \lceil \frac{b+1}{2} \rceil\right)$, $j := j+1$
enddo
$S_{t'}\left(\lfloor \frac{b+1}{2} \rfloor\right) := S_t(b+1)$, $\rho(t) := \lceil \frac{b+1}{2} \rceil$, $\rho(t') := \lfloor \frac{b+1}{2} \rfloor$,
if $i < \rho(u)$ **then** $S_u(\rho(u)+1) := S_u(\rho(u))$ **endif**
$j := \rho(u) - 1$, $\rho(u) := \rho(u)+1$,
while $j > i$ **do**
$S_u(j+1) := S_u(j)$, $H_u(j+1) := H_u(j)$, $j := j-1$
enddo
$S_u(i+1) := t'$, $H_u(i+1) := H_u(i)$, $H_u(i) := H_t(\rho(t))$, $t := u$

DELETE(x)
Vyhledej(x)
if key $(t) = x$ **then**
$u :=$ otec (t), najdi i, že $S_u(i) = t$, a j, že $H_w(j) = x$, $k := i$
if $w \neq u$ a $w \neq NIL$ **then** $H_w(j) := H_u(\rho(u) - 1)$ **endif**
while $k < \rho(u) - 1$ **do**
$H_u(k) := H_u(k+1)$, $S_u(k) := S_u(k+1)$, $k := k+1$
enddo
if $i \neq \rho(u)$ **then** $S_u(\rho(u)-1) := S_u(\rho(u))$ **endif**
$\rho(u) := \rho(u) - 1$, odstraň t, $t := u$

while $\rho(t) < a$ a t není kořen **do**
y je bezprostřední bratr t
if $\rho(y) = a$ **then** **Spojení**(t, y) **else** **Přesun**(t, y) **endif**
enddo
endif

Spojení(t, y)
$u :=$ otec (t), najdi i, že $S_u(i) = t$, $j := 1$
if $S_u(i-1) = y$ **then** vyměň t a y, $i := i - 1$ **endif**
while $j < \rho(y)$ **do**
$S_t(\rho(t) + j) := S_y(j)$, $H_t(\rho(t) + j) := H_y(j)$, $j := j + 1$
enddo
$H_t(\rho(t)) := H_u(i)$, $S_t(\rho(t) + \rho(y)) := S_y(\rho(y))$, $\rho(t) := \rho(t) + \rho(y)$, odstraň y
while $i < \rho(u) - 1$ **do**
$S_u(i+1) := S_u(i+2)$, $H_u(i) := H_u(i+1)$, $i := i + 1$
enddo
$\rho(u) := \rho(u) - 1$
if u je kořen a $\rho(u) = 1$ **then**
odstraň u
else
$t := u$
endif

Přesun(t, y)
$u :=$ otec (t), najdi i takové, že $S_u(i) = t$
if $S_u(i+1) = y$ **then**
$S_t(\rho(t) + 1) := S_y(1)$, $H_t(\rho(t)) := H_u(i)$,
$H_u(i) := H_y(1)$, $j := 1$
while $j < \rho(y) - 1$ **do**
$S_y(j) := S_y(j+1)$, $H_y(j) := H_y(j+1)$, $j := j + 1$
enddo
$S_y(\rho(y) - 1) := S_y(\rho(y))$, $\rho(t) := \rho(t) + 1$, $\rho(y) := \rho(y) - 1$
else
$S_t(\rho(t) + 1) := S_t(\rho(t))$, $j := \rho(t) - 1$
while $j > 0$ **do**
$S_t(j+1) := S_t(j)$, $H_t(j+1) := H_t(j)$, $j := j - 1$
enddo
$\rho(t) := \rho(t) + 1$, $S_t(1) := S_y(\rho(y))$, $H_t(1) := H_u(i-1)$,
$H_u(i-1) := H_y(\rho(y) - 1)$, $\rho(y) := \rho(y) - 1$
endif

MIN
$t :=$kořen stromu
while t není list **do** $t := S_t(1)$ **enddo**
key (t) je nejmenší prvek S

MAX
$t :=$kořen stromu
while t není list **do** $t := S_t(\rho(t))$ **enddo**
key (t) je největší prvek S

JOIN2(T_1, T_2)
Předpoklad T_i je (a, b)-strom reprezentující množinu S_i pro $i = 1, 2$, které splňují $\max S_1 < \min S_2$ (tento předpoklad je silnější než požadavek, že S_1 a S_2 jsou disjunktní, ale algoritmus nekontroluje jeho splnění)
if výška T_1 je větší nebo rovna výšce T_2 **then**
  $t :=$kořen T_1, $k := v(T_1) - v(T_2)$
  while $k > 0$ **do** $t := S_t(\rho(t))$, $k := k - 1$ **enddo**
  Spojení(t, kořen T_2), $t := \text{otec}(t)$
  while $\rho(t) > b$ **do** **Štěpení**(t) **enddo**
else
  $t :=$kořen T_2, $k := v(T_2) - v(T_1)$
  while $k > 0$ **do** $t := S_t(1)$, $k := k - 1$ **enddo**
  Spojení(t, kořen T_1), $t := \text{otec}(t)$
  while $\rho(t) > b$ **do** **Štěpení**(t) **enddo**
endif

SPLIT(T, x)
Z_1, Z_2 prázdné zásobníky, $t :=$kořen T
while t není list **do**
  $i := 1$
  while $H_t(i) < x$ a $i < \rho(t)$ **do** $i := i + 1$ **enddo**
  $t := S_t(i)$
  if $i = 2$ **then** vlož podstrom vrcholu $S_t(1)$ do Z_1 **endif**
  if $i > 2$ **then**
    vytvoř nový vrchol t_1, $\rho(t_1) = i - 1$,
    for every $j = 1, 2, \ldots, i - 2$ **do**
      $S_{t_1}(j) := S_t(j)$, $H_{t_1}(j) := H_t(j)$
    enddo
    $S_{t_1}(i-1) := S_t(i-1)$, vlož podstrom vrcholu t_1 do Z_1
  endif
  if $i = \rho(t) - 1$ **then**
    vlož podstrom $S_t(\rho(t))$ do Z_2
  endif
  if $i < \rho(t) - 1$ **then**
    vytvoř nový vrchol t_2, $\rho(t_2) := \rho(t) - i$
    for every $j = 1, 2, \ldots, \rho(t) - i - 1$ **do**
      $S_{t_2}(j) := S_t(i+j)$, $H_{t_2}(j) := H_t(i+j)$
    enddo
    $S_{t_2}(\rho(t) - i) := S_t(\rho(t))$, vlož podstrom t_2 do Z_2
  endif
enddo
if $\text{key}(t) = x$ **then**
  Výstup: $x \in S$
else
  Výstup: $x \notin S$
  if $\text{key}(t) < x$ **then**
    vlož podstrom vrcholu t do Z_1
  else
    vlož podstrom vrcholu t do Z_2

endif
endif
$T_1 :=$ vrchol Z_1, odstraň T_1 ze Z_1
while $Z_1 \neq \emptyset$ **do**
$T' :=$vrchol Z_1, odstraň T' ze Z_1, $T_1 :=$**JOIN**(T', T_1)
enddo
$T_2 :=$ vrchol Z_2, odstraň T_2 ze Z_2
while $Z_2 \neq \emptyset$ **do**
$T' :=$ vrchol Z_2, odstraň T' ze Z_2, $T_2 :=$**JOIN**(T_2, T')
enddo

Poznámky k algoritmům.
Odkaz na otce vrcholu: buď je v každém vrcholu v stromu T přímo odkaz na otec (v), nebo se v proceduře **Vyhledej** vkládají vrcholy do zásobníku a otec (v) je vrchol v zásobníku před vrcholem v.

Při operaci **SPLIT** se zásobníky používají jednoprůchodově – nejprve se naplní a v této části algoritmu se nepoužije operace **pop**, pak se vyprázdní a v této fázi se nepoužívá operace **push**. V okamžiku, když jsou zásobníky naplněné, platí:

v zásobnících jsou uloženy (a, b)-stromy reprezentující podmnožiny S;
když (a, b)-stromy T_i a T_{i+1} reprezentují množiny S_i a S_{i+1} a jsou v zásobníku Z_1 (nebo Z_2) a strom T_{i+1} následuje po stromu T_i, pak platí $\max S_i < \min S_{i+1} < x$ (nebo $\min S_i > \max S_{i+1} > x$) a výška T_i je větší nebo rovna výšce T_{i+1};
když T_i a T_{i+1} jsou dva po sobě následující (a, b)-stromy v zásobníku Z_j pro $j = 1, 2$, které mají stejnou výšku, pak následující strom v zásobníku Z_j má ostře menší výšku.

Toto plyne z první fáze algoritmu operace **SPLIT** a zajišťuje korektnost druhé fáze algoritmu.

Dále si všimněme, že podprocedury **Štěpení**, **Spojení** a **Přesun** vyžadují čas $O(1)$, a proto algoritmy pro operace **MEMBER**, **INSERT**, **DELETE**, **MIN**, **MAX**, **JOIN2** a pro první fázi algoritmu **SPLIT** vyžadují čas $O(1)$ pro práci v dané hladině. Protože hladin je nejvýše $\log_a |S|$, můžeme shrnout:

Věta. *Algoritmy pro operace* **MEMBER**, **INSERT**, **DELETE**, **MIN**, **MAX**, **JOIN2** *a* **SPLIT** *v (a, b)-stromech vyžadují v nejhorším případě čas $O(\log_a |S|)$, kde S je reprezentovaná množina.*

Je třeba ještě odhadnout spotřebovaný čas ve druhé fázi algoritmu pro operaci **SPLIT**. Nejprve si všimněme, že algoritmus **JOIN2**(T_1, T_2) vyžaduje ve skutečnosti jen čas rovný O (rozdíl výšek stromů T_1 a T_2). Když po naplnění zásobník Z_j pro $j = 1, 2$ obsahuje stromy $U_1, U_2, \ldots, U_k$ v tomto pořadí, pak $k \leq 2 \log_a |S|$ a vyprázdnění zásobníku Z_j vyžaduje čas $O\left(\sum_{i=1}^{k-1} (u_i - u_{i+1} + 1)\right) = O(u_1 + k)$, kde u_i je výška stromu U_i pro $i = 1, 2, \ldots, k$. Protože výška stromu U_1 je nejvýše rovna výšce stromu T, dostáváme, že druhá fáze algoritmu **SPLIT** vyžaduje čas $O(\log_a |S|)$ a důkaz je kompletní.

Nyní popíšeme algoritmus pro operaci **ord**(k). Tato operace se často nazývá k-tá pořádková statistika Tato operace není podporována navrženou strukturou, pro její efektivní implementaci musíme rozšířit strukturu vnitřního vrcholu v o pole
$P_v(1..\rho(v) - 1)$, kde $P_v(i)$ je počet prvků S reprezentovaných v podstromu i-tého syna vrcholu v.

Udržovat pole P_v v aktuálním stavu znamená při úspěšném provedení aktualizační operace projít cestu z vrcholu do kořene a aktualizovat pole P. Uvedeme algoritmus pro nalezení k-té pořádkové statistiky.

ord(k)
if $k > |S|$ **then** neexistuje k-tý nejmenší prvek, konec **endif**
t :=kořen stromu
while t není list **do**
  $i := 1$
  while $k > P_t(i)$ a $i < \rho(t)$ **do**
    $k := k - P_t(i)$, $i := i + 1$
  enddo
  $t := S_t(i)$
enddo
key (t) je hledaný k-tý nejmenší prvek

Invariant algoritmu: V každém okamžiku platí, že původní k se rovná aktuální k+počet prvků z S, které jsou v podstromech vrcholů stromu, které v lexikografickém uspořádání předcházejí i-tému synu vrcholu t. Korektnost algoritmu plyne z tohoto invariantu.

Věta. *Algoritmy pro operace* **MEMBER**, **INSERT**, **DELETE**, **MIN**, **MAX**, **SPLIT**, **JOIN2** *a* **ord**(k) *pro všechna k v rozšířené struktuře (a,b)-stromu vyžadují v nejhorším případě čas $O(\log|S|)$, kde S je reprezentovaná množina.*

(a,b)-stromy se používají jak v interní tak v externí paměti. Jaké hodnoty a a b je vhodné používat?
Pro interní paměť jsou doporučené hodnoty $a = 2$, $b = 4$ nebo $a = 3$ a $b = 6$.
Pro externí paměť jsou doporučené hodnoty $a \approx 100$, $b = 2a$.

Když je množina reprezentovaná (a,b)-stromem uložena na serveru a má k ní přístup více uživatelů, vzniká problém s aktualizačními operacemi. Tyto operace mění strukturu (a,b)-stromu a v důsledku toho se v něm jiný uživatel může ztratit. Tento problém se dá řešit tak, že při aktualizačních operacích se uzavře celý strom.

Nevýhoda: ostatní uživatelé do něho nemají přístup a nemohou pracovat. Tzv. paralelní implementace operací **INSERT** a **DELETE** nabízí jiné, efektivnější řešení.

Předpoklad: $b \geq 2a$.
Při operaci **INSERT** jsou ve vyhledávací fázi vždy uzavřeny vrcholy t, otec (t) a synové vrcholu t. Algoritmus zjistí, ve kterém synu vrcholu t má pokračovat, a pak, když $\rho(t) = b$, provede **Štěpení** (proto je nutně $b \geq 2a$, abychom po této operaci měli zase (a,b)-strom). V algoritmu pak odpadne vyvažovací část (tj. **Štěpení** při cestě vzhůru ke kořeni).

Při operaci **DELETE** jsou ve vyhledávací fázi uzavřeny vrcholy t, otec (t), bezprostřední bratr y vrcholu t a jejich synové. Když $\rho(t) = a$, pak po najiti vrcholu, kde se bude pokračovat, se provede buď **Přesun** (když $\rho(y) > a$) nebo **Spojení** (když $\rho(y) = a$). Stejně jako při operaci **INSERT** se vynechá vyvažovací část uzavírající původní algoritmus.

Tato úprava vyžaduje sice více **Štěpení**, **Spojení** a **Přesunů**, ale asymptoticky vychází čas stejný (jen je větší multiplikativní konstanta). Doporučené hodnoty a a b jsou $a \approx 100$ a $b = 2a+2$ při uložení na serveru v externí paměti, ve vnitřní paměti se doporučuje $a = 2$, $b = 6$.

Operace **JOIN2** lze také paralelizovat, ale operaci **SPLIT** paralelizovat nelze.

(a, b)-stromy dávají také zajímavé aplikace pro třídicí algoritmy. Použití (a, b)-stromů pro setřídění náhodné posloupnosti není vhodné, režie na udržování struktury (a, b)-stromu vede k tomu, že multiplikativní konstanta by byla o hodně větší než u klasických třídicích algoritmů. Také uložení (a, b)-stromu vyžaduje více paměti než je potřeba pro klasické algoritmy. Situace se podstatně změní, když vstupní posloupnost je předtříděná a je ji třeba jen dotřídit. Klasické algoritmy většinou nejsou schopné využít faktu, že posloupnost je předtříděná, a jejich časová náročnost je prakticky stejná (někdy i horší) jako u náhodné posloupnosti. Na rozdíl od nich algoritmus A-**sort** založený na (a, b)-stromech je schopen předtříděnost využít a má na předtříděných posloupnostech lepší výsledky než klasické algoritmy.

Modifikace (a, b)-stromů pro algoritmus A-**sort**. Máme (a, b)-strom reprezentující vstupní posloupnost, je dán ukazatel Prv na první list, listy (a, b)-stromu jsou propojeny do seznamu v rostoucím lexikografickém pořadí (ukazatel na následující prvek je Nasl) a je dána cesta z prvního listu do kořene (to znamená, že na cestě z prvního listu do kořene známe pro každý vrchol v jeho otce). Nyní uvedeme algoritmus A-**sort**.

  A-**sort**$(x_1, x_2, \dots, x_n)$
$i := n - 1$, vytvoř jednoprvkovy strom s vrcholem t
key $(t) := x_n$, Prv $:= t$
while $i \geq 1$ **do** A-**Insert**(x_i), $i := i - 1$ **enddo**
$y_1 :=$ key (Prv)
while $i \leq n$ **do**
  $y_i :=$ key (t), $i := i + 1$, $t :=$ Nasl (t)
enddo
Výstup: $(y_1, y_2, \dots, y_n)$ setříděná posloupnost $(x_1, x_2, \dots, x_n)$

  A-**Insert**(x)
$t :=$ Prv
while $t \neq$ kořen T a $H_t(1) < x$ **do** $t :=$ otec (t) **enddo**
while $t \neq$ list **do**
  $i := 1$
  while $H_t(i) < x$ a $i < \rho(t)$ **do** $i := i + 1$ **enddo**
  if $i > 1$ **then** $v := S_t(i-1)$ **else** $v := S_t(\rho(t))$ **endif**
  $t := S_t(i)$
enddo
if key $(t) \neq x$ **then**
  vytvoř nový list t', key $(t') = x$,
  if t je kořen **then**
    vytvoř nový kořen u, $\rho(u) := 2$
    if key $(t) > x$ **then**
      $H_u(1) := x$, $S_u(1) := t'$, $S_u(2) := t$,
      Prv $:= t'$, Nasl $(t') := t$, Nasl $(t) := NIL$
    else
      $H_u(1) :=$ key (t), $S_u(1) := t$, $S_u(2) := t'$
      Prv $:= t$, Nasl $(t) := t'$, Nasl $(t') := NIL$
    endif
  else
    $u :=$ otec (t)

if key $(t) < x$ **then**
(komentář: $x > \max S$)
$S_u(\rho(u)+1) := t'$, $H_u(\rho(u)) :=$ key (t), $\rho(u) := \rho(u)+1$
Nasl $(t) := t'$, Nasl $(t') := NIL$
else
najdi i, že $S_u(i) = t$, $S_u(\rho(u)+1) := S(\rho(u))$,
$j := \rho(u) - 1$, Nasl $(v) := t'$, Nasl $(t') := t$
while $j \geq i$ **do**
$S_u(j+1) := S_u(j)$, $H_u(j+1) := H_u(j)$, $j := j-1$
enddo
$S_u(i) := t'$, $H_u(i) := x$, $\rho(u) := \rho(u)+1$,
if $t =$ Prv **then** Prv $:= t'$ **endif**
endif
$t := u$
while $\rho(t) > b$ **do** **Štěpení**(t) **enddo**
endif
endif

Korektnost algoritmu plyne z faktu, že key je izomorfismus uspořádání a seznam listů je v rostoucím pořadí. Protože v je vždy bezprostřední předchůdce t, je seznam korektně definován. Ukazatel otec (t) je dán na cestě z vrcholu Prv do kořene, pro ostatní vrcholy se řeší stejným způsobem jako pro (a,b)-stromy.

Složitost algoritmu: Algoritmus A-**sort** vyžaduje více času i více paměti než klasické třídicí algoritmy, ale jejich asymptotická složitost je stejná. Jeho výhoda je v použití na předtříděné posloupnosti. Mějme posloupnost $(x_1, x_2, \ldots, x_n)$ prvků z totálně uspořádaného univerza U a definujme
$$F = |\{(i,j) \mid i < j,\, x_j < x_i\}|.$$

Zřejmě $F = 0$, právě když posloupnost $(x_1, x_2, \ldots, x_n)$ je setříděná. Dále $0 \leq F \leq \binom{n}{2}$ a $F = \binom{n}{2}$, právě když je posloupnost $(x_1, x_2, \ldots, x_n)$ klesající. To vede k tomu brát F jako míru předtříděnosti posloupnosti. Spočítáme složitost algoritmu A-**sort** v závislosti na n a F

Zřejmě algoritmus A-**sort v** nejhorším případě vyžaduje čas, který potřebuje A-**Insert**, plus $O(n)$. Algoritmus A-**Insert**(x) vyžaduje čas potřebný na nalezení místa, kam vložit x, plus $O\big($počet volání **Štěpení**$\big)$. Protože každý běh procedury **Štěpení** vytvořil jeden vnitřní vrchol (a,b)-stromu a protože $a \geq 2$ a (a,b)-strom po skončení volání A-**Insert** má n listů, je vnitřních vrcholů (a,b)-stromu $< n$. Proto všechny běhy procedury A-**Insert** vyžadují čas na nalezení míst jednotlivých prvků plus $O(n)$. Když procedura A-**Insert**(x) při hledání místa pro prvek x skončila ve výšce h (tj. první cyklus se h-krát opakoval), pak nalezení místa pro prvek x vyžadovalo čas $O(h)$. Všechny prvky reprezentované (a,b)-stromem pod prvním vrcholem ve výšce $h-1$ jsou menší než x a je jich alespoň a^{h-1}. Když $x = x_i$, pak počet prvků reprezentovaných (a,b)-stromem při běhu procedury A-**Insert**(x), které jsou menší než x, je počet j takových, že $i < j$ a $x_j < x_i$. Označme f_i tento počet. Pak platí

$$a^{h-1} \leq f_i \implies h - 1 \leq \log_a f_i \implies h \in O(\log f_i).$$

Proto v nejhorším případě čas potřebný pro nalezení pozice x_i je $O(\log f_i)$. Odtud plyne, že čas algoritmu potřebný k běhu algoritmu A-**sort** je

$$O\left(\left(\sum_{i=1}^{n}\log f_i\right)+n\right).$$

Zřejmě $\sum_{i=1}^{n} f_i = F$ a nyní využijeme toho, že geometrický průměr je vždy menší nebo roven aritmetickému průměru, a odtud dostáváme

$$\sum_{i=1}^{n}\log f_i = \log\prod_{i=1}^{n} f_i = n\log\left(\prod_{i=1}^{n} f_i\right)^{\frac{1}{n}} \leq$$
$$n\log\frac{\sum_{i=1}^{n} f_i}{n} = n\log\frac{F}{n}.$$

Věta. *Algoritmus A-**sort** na setřídění n-členné posloupnosti vyžaduje v nejhorším případě čas $O\left(n + n\log\frac{F}{n}\right)$, kde F je míra setříděnosti vstupní posloupnosti.*

Zhodnocení: Protože A-**sort** nepoužívá operaci **DELETE**, doporučuje se použít $(2,3)$-stromy. Když se budou třídit posloupnosti s mírou $F \leq n\log n$, pak algoritmus A-**sort** bude potřebovat v nejhorším případě čas $O(n\log\log n)$. Mehlhorn a Tsakalidis dokázali, že když $F \leq 0.02n^{1.57}$, pak algoritmus A-**sort** je rychlejší než algoritmus **Quicksort**.

Propojené stromy s prstem.
<u>Hladinově</u> <u>propojený</u> (a,b)-<u>strom</u> <u>s</u> <u>prstem</u> je (a,b)-strom, kde struktura vnitřního vrcholu různého od kořene je rozšířena (proti klasickému (a,b)-stromu) o ukazatele:
otec (v), levy (v), pravy (v), kde
levy (v) ukazuje na největší vrchol (v lexikografickém uspořádání) ve stejné hladině jako v, který je menší než v (když neexistuje, tak je to NIL),
pravy (v) ukazuje na nejmenší vrchol (v lexikografickém uspořádání) ve stejné hladině jako v, který je větší než v (když neexistuje, tak je to NIL).
Navíc je dán ukazatel Prst na některý list.

Zde se liší hlavně vyhledávání, které je zobecněním postupu A-**sortu**. Začínáme od listu p, na který ukazuje Prst. Když x je menší než prvek reprezentovaný tímto listem, pak se pokračuje v jeho otci v, a když p byl i-tý syn v, tak se pomocí pole H_v zjišťuje, zda x nemá být reprezentován v podstromu jeho j-tého syna pro $j < i$. Když ne, pokračuje se ukazatelem levy (v). Když x není reprezentován ani v jeho podstromu, tak se celý postup opakuje o hladinu výš (zkoumá se otec vrcholu). Když x je větší než prvek reprezentovaný listem p, je postup zrcadlově obrácený. Když se nalezne vrchol, v jehož podstromu má x ležet, pak se aplikuje od tohoto vrcholu (místo od kořene) procedura **Vyhledej**.

Struktura kromě operací uspořádaného slovníkového problému ještě používá přidanou operaci **PRST**(x), která nastaví ukazatel Prst na list, který reprezentuje nejmenší prvek větší nebo rovný x (pokud $x > \max S$, tak ukazatel Prst bude ukazovat na největší list). Operace provedou vyhledání a pak pokračují klasickým způsobem.

Použití: Tato struktura je velmi výhodná pro úlohy, kde vždy skupina po sobě jdoucích operací pracuje v blízkém okolí nějakého $x \in U$. Pak vyhledání prvku je rychlejší než v klasickém (a, b)-stromu, viz A-**sort**.

Vyvažovací operace **Štěpení, Spojování, Přesun** vyžadují čas $O(1)$, ale ve skutečnosti jsou nejpomalejší částí algoritmů pro operace **INSERT** a **DELETE**. Omezení jejich počtu vedlo k menší složitosti algoritmu A-**sort**. To motivovalo analýzu jejich použití.
Libovolný běh algoritmu **INSERT** volá podproceduru **Štěpení** nejvýše $\log(|S|)$-krát a libovolný běh algoritmu **DELETE** může nejvýše $\log(|S|)$-krát zavolat podproceduru **Spojení** a nejvýše jednou podproceduru **Přesun**. V obecném případě tyto odhady nejdou zlepšit. Pro vhodný typ (a, b)-stromu však amortizovaný počet vyvažovacích operací (začínáme-li s původně prázdným stromem) je konstantní.

Pro pevné a a b označme

$$c = \min\left\{\min\left\{2a-1, \left\lceil\frac{b+1}{2}\right\rceil\right\} - a, b - \max\left\{2a-1, \lfloor\frac{b+1}{2}\rfloor\right\}\right\}.$$

Připomínáme, že <u>výška</u> <u>vrcholu</u> v kořenovém stromě je maximální délka cesty z něho do některého listu.

Věta. *Nechť $b \geq 2a$ a $a \geq 2$. Nechť $\mathcal{P}$ je posloupnost n operací* **INSERT** *a* **DELETE**, *aplikujme ji na prázdný (a, b)-strom. Označme*
St_h – počet **Štěpení** *ve výšce h při aplikaci $\mathcal{P}$, $St = \sum_h St_h$;*
Sp_h – počet **Spojení** *ve výšce h při aplikaci $\mathcal{P}$, $Sp = \sum_h Sp_h$;*
P_h – počet **Přesunů** *ve výšce h při aplikaci $\mathcal{P}$, $P = \sum_h P_h$.*
Pak platí

(1)
$$P \leq n \quad \text{a} \quad (2c-1)\,St + cSp \leq n + c + \frac{c\,(n-2)}{a+c-1};$$

(2)
$$St_h + Sp_h + P_h \leq \frac{2\,(c+2)\,n}{(c+1)^h}.$$

Z definice plyne, že $c \geq 1$, a protože $a \geq 2$, z 1) dostaneme

$$St + Sp \leq \frac{n}{c} + 1 + \frac{n-2}{a} \leq n + 1 + \frac{n-2}{2} \leq \frac{3n}{2}.$$

Amortizovaný počet vyvažovacích operací splňuje tedy

$$\frac{P + St + Sp}{n} \leq \frac{5}{2}.$$

Důkaz je založen na bankovním principu – navrhneme kvantitativní ohodnocení (a, b)-stromu, nalezneme jeho horní odhad a popíšeme, jak toto ohodnocení mohou změnit vyvažovací operace. Srovnání těchto odhadů dá požadovaný výsledek.

Mějme (a,b)-strom T, pro vnitřní vrchol v různý od kořene definujme

$$b(v) = \min\{\rho(v) - a, b - \rho(v), c\},$$

pro kořen r definujme

$$b(r) = \min\{\rho(r) - 2, b - \rho(r), c\}.$$

Pozorování. *Pro vnitřní vrchol stromu v různý od kořene platí*

(1) $b(v) \leq c$;
(2) *když* $\rho(v) = a$ *nebo* $\rho(v) = b$, *pak* $b(v) = 0$;
(3) *když* $\rho(v) = a - 1$ *nebo* $\rho(v) = b + 1$, *pak* $b(v) = -1$;
(4) *když* $\rho(v) = 2a - 1$, *pak* $b(v) = c$;
(5) *Když v' a v'' jsou dva různé vrcholy stromu různé od kořene takové, že* $\rho(v') = \lceil \frac{b+1}{2} \rceil$ *a* $\rho(v'') = \lfloor \frac{b+1}{2} \rfloor$, *pak* $b(v') + b(v'') \geq 2c - 1$;
(6) *pro kořen r platí* $b(r) \leq c$.

Strom (T, r) ohodnotíme

$$b_h(T) = \sum \{b(v) \mid v \neq r \text{ vnitřní vrchol stromu ve výšce } h\}$$
$$b(T) = \sum_{h=1}^{\infty} b_h(T) + b(r).$$

Řekneme, že (T, r, v) je <u>parciální</u> (a,b)-<u>strom</u>, když r je kořen stromu, v je vnitřní vrchol T a platí:

když $v \neq r$, pak $a - 1 \leq \rho(v) \leq b + 1$ a $2 \leq \rho(r) \leq b$;
když $v = r$, pak $2 \leq \rho(r) \leq b + 1$;
když t je vnitřní vrchol T různý od v a r, pak

$$a \leq \rho(t) \leq b;$$

všechny cesty z kořene r do nějakého listu mají stejnou délku.

Nyní rozložíme operace **INSERT** a **DELETE** do jednotlivých akcí se stromem a vyšetříme vliv těchto akcí na jeho ohodnocení. Důkazy lemmat jsou založené na následujícím pozorování

Pozorování. *Mějme dva stromy T a T', které mají stejnou množinu vrcholů ve výšce h. Pak platí:*

(1) *když každý vrchol ve výšce h má stejný počet synů v obou stromech, pak* $b_h(T) = b_h(T')$;
(2) *když všechny vrcholy ve výšce h až na jeden vrchol mají stejný počet synů v obou stromech a počet synů u zbylého vrcholu se ve stromech T a T' liší nejvýše o 1, pak* $b_h(T) \geq b_h(T') - 1$.

Lemma 1. *Když (T, r) je (a,b)-strom a když strom T' vznikne z T přidáním/ubráním jednoho syna vrcholu v ve výšce 1 (tj. přidávaný/ubíraný syn je list), pak (T', r, v) je parciální (a,b)-strom a platí*

$$b_1(T') \geq b_1(T) - 1 \quad \text{a } b_h(T') = b_h(T) \text{ pro } h > 1;$$
$$b(T') \geq b(T) - 1.$$

Lemma 2. *Nechť* (T, r, v) *je parciální* (a, b)*-strom,* $\rho(v) = b + 1$ *a* v *je ve výšce* $l \geq 1$*. Když* T' *vznikne z* T *operací* **Štěpení**(v)*, pak* $(T', r, \mathrm{otec}(v))$ *je parciální* (a, b)*-strom a platí:*

$$\begin{aligned} b_l(T') \geq& b_l(T) + 2c, \; b_{l+1}(T') \geq b_{l+1}(T) - 1 \\ b_h(T') =& b_h(T) \text{ pro } h \neq l, l+1; \quad b(T') \geq b(T) + 2c - 1. \end{aligned}$$

Lemma 3. *Nechť* (T, r, v) *je parciální* (a, b)*-strom,* $\rho(v) = a - 1$*,* v *je ve výšce* $l \geq 1$ *a* y *je bezprostřední bratr* v *takový že* $\rho(y) = a$*. Když* T' *vznikne z* T *operací* **Spojení**(v, y)*, pak* $(T', r, \mathrm{otec}(v))$ *je parciální* (a, b)*-strom a platí:*

$$\begin{aligned} b_l(T') \geq& b_l(T) + c + 1, \; b_{l+1}(T') \geq b_{l+1}(T) - 1 \\ b_h(T') =& b_h(T) \text{ pro } h \neq l, l+1; \quad b(T') \geq b(T) + c. \end{aligned}$$

Lemma 4. *Nechť* (T, r, v) *je parciální* (a, b)*-strom,* $\rho(v) = a - 1$*,* v *je výšce* $l \geq 1$ *a* y *je bezprostřední bratr* v *takový, že* $\rho(y) > a$*. Když* T' *vznikne z* T *operací* **Přesun**(v, y)*, pak* (T', r) *je* (a, b)*-strom a platí:*

$$b_l(T') \geq b_l(T) \text{ a } b_h(T') = b_h(T) \text{ pro } h \neq l; \quad b(T') \geq b(T).$$

Označme T_k (a, b)-strom vzniklý provedením posloupnosti $\mathcal{P}$ na prázdný (a, b)-strom. Sečtením předchozích výsledků dostáváme

Důsledek 5. *Když položíme*

$$\begin{gathered} St_0 + Sp_0 = \text{počet listů v } T_k \leq n, \quad \text{pak} \\ b_h(T_k) \geq 2cSt_h + (c+1) Sp_h - St_{h-1} - Sp_{h-1} \text{ pro } h \geq 1. \end{gathered}$$

Dále $b(T_k) \geq (2c - 1) St + cSp - n$*, kde* n *je délka posloupnosti* $\mathcal{P}$*.*

První výraz upravíme (využíváme, že $c \geq 1$):

$$\begin{aligned} St_h + Sp_h \leq& \frac{b_h(T_k)}{c+1} + \frac{St_{h-1} + Sp_{h-1}}{c+1} \leq \\ & \frac{b_h(T_k)}{c+1} + \frac{b_{h-1}(T_k)}{(c+1)^2} + \frac{St_{h-2} + Sp_{h-2}}{(c+1)^2} \leq \cdots \leq \\ & \sum_{i=0}^{h-1} \frac{b_{h-i}(T_k)}{(c+1)^{i+1}} + \frac{n}{(c+1)^h} = \\ & \frac{n}{(c+1)^h} + \sum_{l=1}^{h} b_l(T_k) \frac{(c+1)^l}{(c+1)^{h+1}}. \end{aligned}$$

Nyní odhadneme shora $b(T_k)$.

Lemma 6. *Když T je (a,b)-strom s m listy, pak* $0 \le b(T) \le c + (m-2)\frac{c}{a+c-1}$.

Důkaz. Pro $0 \le j < c$ označme m_j počet vnitřních vrcholů různých od kořene, které mají přesně $a+j$ synů, a m_c označme počet vnitřních vrcholů různých od kořene, které mají alespoň $a+c$ synů. Když vrchol v má $a+j$ synů, pak $b_T(v) \le j$ a pro každý vnitřní vrchol v platí $b_T(v) \le c$. Tedy $b(T) \le c + \sum_{j=0}^{c} jm_j$. Z vlastností stromů plyne

$$2 + \sum_{j=0}^{c}(a+j)\,m_j \le \sum\{\rho(v) \mid v \text{ je vnitřní vrchol } T\} =$$

$$m + \sum_{j=0}^{c} m_j.$$

Odtud plyne

$$\sum_{j=0}^{c}(a+j-1)\,m_j \le m-2.$$

Protože $\frac{j}{a+j-1} \le \frac{c}{a+c-1}$ pro každé j takové, že $0 \le j \le c$, dostáváme

$$b(T) \le c + \sum_{j=0}^{c} jm_j = c + \sum_{j=0}^{c}\frac{j}{a+j-1}(a+j-1)\,m_j \le$$

$$c + \frac{c}{a+c-1}(m-2)$$

a lemma je dokázáno. $\square$

Nyní dokážeme tvrzení (1) Věty. Protože každá operace **DELETE** použije nejvýše jednu operaci **Přesun** (a operace **INSERT** operaci **Přesun** nepoužívá) dostáváme, že

$$P \le \text{počet operací } \textbf{DELETE} \le n$$

a první nerovnost platí. Abychom dokázali druhou nerovnost, spojíme druhé tvrzení v Důsledku 5 a Lemma 6 (T_k má nejvýše n listů)

$$(2c-1)\,St + cSp - n \le b(T_k) \le c + (n-2)\frac{c}{a+c-1}$$

Odtud plyne požadovaná nerovnost a (1) je dokázáno.

Důkaz (2) využije následující odhad.

Lemma 7. *Pro každé $h \ge 1$ a pro každý (a,b)-strom T s m listy platí*

$$\sum_{l=1}^{h} b_l(T)\,(c+1)^l \le (c+1)\,m.$$

Důkaz. Pro $0 \le j < c$ a pro libovolné h označme $m_j(h)$ počet vrcholů ve výšce h různých od kořene, které mají přesně $a+j$ synů, a $m_c(h)$ počet vrcholů ve výšce h různých od kořene, které mají alespoň $a+c$ synů. Pak máme

$$b_h(T) \le \sum_{j=0}^{c} jm_j(h),$$

$$\sum_{j=0}^{c}(a+j)\,m_j(h) \le \sum_{j=0}^{c} m_j(h-1) \text{ pro každé } h \ge 1,$$

kde dodefinováváme $\sum_{j=0}^{c} m_j(0) = m$. Tyto vztahy použijeme v následujícím odhadu. Platí

$$\sum_{l=1}^{h} b_l(T)(c+1)^l \leq \sum_{l=1}^{h}\left[(c+1)^l\left(\sum_{j=0}^{c} j m_j(l)\right)\right] \leq$$
$$\sum_{l=1}^{h}\left[(c+1)^l\left(\sum_{j=0}^{c} m_j(l-1) - a\sum_{j=0}^{c} m_j(l)\right)\right] =$$
$$(c+1)\sum_{j=0}^{c} m_j(0) - (c+1)^h a \sum_{j=0}^{c} m_j(h) +$$
$$\sum_{l=1}^{h-1}(c+1)^{l+1}\left(\sum_{j=0}^{c} m_j(l) - \frac{a}{c+1}\sum_{j=0}^{c} m_j(l)\right) \leq$$
$$(c+1)m,$$

kde rovnost jsme získali přerovnáním sčítanců tak, aby výrazy $\sum_{j=0}^{c} m_j(l)$ byly u sebe, a poslední nerovnost plyne z toho, že $\frac{a}{c+1} \geq 1$, a tedy třetí sčítanec v předchozím výrazu není kladný. □

Zkombinujeme odhad $St_h + Sp_h$ s Lemmatem 7 a dostaneme

$$St_h + Sp_h \leq \frac{n}{(c+1)^h} + \sum_{l=1}^{h} b_l(T_k)\frac{(c+1)^l}{(c+1)^{h+1}} \leq$$
$$\frac{n}{(c+1)^h} + \frac{n(c+1)}{(c+1)^{h+1}} = \frac{2n}{(c+1)^h}.$$

Protože $P_h \leq Sp_{h-1} - Sp_h \leq St_{h-1} + Sp_{h-1} \leq \frac{2n}{(c+1)^{h-1}}$ dostáváme, že

$$St_h + Sp_h + P_h \leq \frac{2n}{(c+1)^h} + \frac{2n}{(c+1)^{h-1}} = \frac{2n + 2n(c+1)}{(c+1)^h} =$$
$$\frac{2n(c+2)}{(c+1)^h}$$

a důkaz (2) ve Větě je hotov.

Věta vysvětluje, proč jsou doporučené hodnoty $b \geq 2a$ – pak je počet vyvažovacích operací během posloupnosti operací **INSERT** a **DELETE** lineární vzhledem k délce této posloupnosti. Pro $b = 2a - 1$ lze lehce nalézt posloupnost operací **INSERT** a **DELETE** o délce n takovou, že její aplikace na prázdný (a, b)-strom vyžaduje počet vyvažovacích operací úměrný $n \log n$ (pro každé dostatečně velké n). Podobná věta platí i pro paralelní implementaci (a, b)-stromů, ale platí za předpokladu $b \geq 2a+2$. Pro $b = 2a$ nebo $b = 2a+1$ lze nalézt posloupnost, která je protipříkladem. Proto se doporučuje hodnota $b = 2a + 2$ pro paralelní implementaci (a, b)-stromu. Pro propojené (a, b)-stromy platí silnější verze.

Věta. *Předpokládejme, že $b \geq 2a$ a $a \geq 2$. Mějme hladinově propojený (a, b)-strom s prstem T, který reprezentuje n-prvkovou množinu. Pak posloupnost $\mathcal{P}$ operací* **MEMBER**, **INSERT**, **DELETE** *a* **PRST** *aplikovaná na T vyžaduje čas*

$$O(\log(n) + \text{ čas na vyhledání prvků}).$$

Vysvětlení: Začínáme v libovolném propojeném (a, b)-stromě T, proto jeho struktura může být nevýhodná pro danou posloupnost operací $\mathcal{P}$. Abychom se dostali do vhodného režimu, může být třeba až $\log(n)$ vyvažovacích operací. Čas na vyhledávání nemůžeme ovlivnit, ten musí ovlivnit uživatel.

Aplikace: analýza hladinově propojených stromů s prstem umožnila návrh algoritmu, který pro dvě množiny S_1 a S_2 reprezentované propojenými (a, b)-stromy, kde $b \geq 2a$ a $a \geq 2$, zkonstruuje propojený (a, b)-strom reprezentující množinu $S_1 \cup S_2$ (nebo množinu $\Delta(S_1, S_2) = (S_1 \setminus S_2) \cup (S_2 \setminus S_1)$ nebo $S_1 \cap S_2$ nebo $S_1 \setminus S_2$) v čase $O\left(\log \binom{n+m}{m}\right)$, kde $n = \max\{|S_1|, |S_2|\}$ a $m = \min\{|S_1|, |S_2|\}$. Detaily budou v letním semestru.

Vyvažování při operaci **INSERT** lze provádět tak, že operace **Štěpení**(t) se provede, jen když oba bratři vrcholu t mají b synů. Jinak se provádí operace **Přesun**. Nevím o žádném seriózním pokusu tyto alternativy porovnat.

Vyhledávání v uspořádaném poli

Zadání úlohy: Máme podmnožinu S lineárně uspořádaného univerza a S je uložena v poli $A[1..|S|]$ tak, že pro $i < j$ je $A(i) < A(j)$. Pro dané $x \in U$ máme zjistit, zda $x \in S$ (operace **MEMBER**(x)).

Řešení: Pokud $x < A(1)$ nebo $A(|S|) < x$, pak x není prvkem S. V opačném případě buď $x = A(1)$ nebo $x = A(|S|)$ nebo máme dvě hodnoty d a h takové, že $1 \leq d < d+1 < h \leq |S|$ a $A(d) < x < A(h)$. Pak najdeme n takové, že $d < n < h$, a dotazem zjistíme, zda $x = A(n)$ (pak končíme a $x \in S$) nebo $x < A(n)$ (pak položíme $h = n$) nebo $x > A(n)$ (pak položíme $d = n$) a proces opakujeme. Končíme, když $d + 1 \geq h$, pak $x \notin S$. Na začátku položíme $d = 1$ a $h = |S|$. Formální zápis algoritmu:

```
  MEMBER(x)
if x = A(1) then
  Výstup: x ∈ S stop
else
  if x < A(1) then
    Výstup: x ∉ S stop
  else
    d = 1
  endif
endif
if x = A(|S|) then
  Výstup: x ∈ S stop
else
  if x > A(|S|) then
    Výstup: x ∉ S stop
  else
    h = |S|
  endif
endif
while d + 1 < h do
  n := next(d, h)
  if x = A(n) then
    Výstup: x ∈ S stop
```

else
if $x < A(n)$ **then** $h := n$ **else** $d := n$ **endif**
endif
enddo
Výstup: $x \notin S$ stop

V tomto metaalgoritmu je **next**(d, h) funkce, která nalezne hodnotu n takovou, že $d < n < h$. Korektnost plyne z pozorování, že když $d + 1 = h$, pak $A(d) < x < A(h)$ implikuje, že neexistuje i takové, že $x = A(i)$, a tedy $x \notin S$. Efektivita algoritmu záleží na fukci **next**. Zpracování dotazu vyžaduje čas $O(1)$ a počet dotazů je počet volání funkce **next**.

Unární vyhledávání: **next**$(d, h) = d + 1$, pak každý dotaz zvětší d o 1, a tedy největší počet dotazů je $|S|$. Algoritmus v nejhorším případě vyžaduje čas $O(|S|)$ a očekávaný počet dotazů při rovnoměrném rozložení množiny S a prvku x je $\frac{|S|}{2}$ (tedy očekávaný čas je $O(|S|)$).

Poznámka: Duální přístup je, když **next**$(d, h) = h - 1$, výsledky se nezmění. Při aplikacích je někdy výhodné použít funkci **next**$(d, h) = \min\{d + c, h - 1\}$, kde c je nějaká konstanta (pak krok není 1, ale c). Jak uvidíme později, jsou situace, kdy je výhodné takovéto unární vyhledávání použít.

Binární vyhledávání: **next**$(d, h) = \left\lceil \frac{d+h}{2} \right\rceil$, pak každý dotaz zmenší rozdíl $h - d$ přibližně na polovinu. Počet dotazů je nejvýše $3 + \log(|S| - 2)$, algoritmus tedy v nejhorším případě vyžaduje čas $O(\log|S|)$ a očekávaný čas při rovnoměrném rozložení množiny S a $x \in U$ je také $O(\log|S|)$.

Interpolační vyhledávání: **next**$(d, h) = d + \left\lceil \frac{x - A(d)}{A(h) - A(d)}(h - d) \right\rceil$. V nejhorším případě musíme položit více než $\frac{|S|}{2}$ dotazů, a proto čas v nejhorším případě je $O(|S|)$, ale při rovnoměrném rozložení množiny S a $x \in U$ je očekávaný čas $O(\log\log|S|)$. To je založeno na faktu, že hodnota **next** závisí i na velikosti x. Když x je velké, tak hodnota **next** je posunuta do větších hodnot, když x je malé, pak je posunuta do menších hodnot.

Poznámka: Když rozložení prvků není rovnoměrné, ale je známé, pak podle toho můžeme upravit funkci **next** a očekávaný čas algoritmu se nezmění.

Pro následující funkci **next** bude jednodušší spočítat očekávaný počet dotazů než pro interpolační vyhledávání, ale výsledek je asymptoticky stejný.

Zobecněné kvadratické vyhledávání.
Funkce **next** je zde definována složitější procedurou, jejíž výsledek závisí i na předchozích situacích a na výsledku dotazu. Procedura zadává dotazy v blocích. První dotaz v bloku je interpolační a procedura přitom zjistí velikost kroku a zda x je menší nebo větší než první dotaz v bloku. Pak střídá unární a binární vyhledávání. Blok končí, když rozdíl mezi h a d je nejvýše velikost kroku. Krok v následujícím bloku klesne přibližně na odmocninu velikosti kroku v tomto bloku. Procedura používá boolské proměnné *blok*, *typ*, *smer*. Proměnná *blok* je inicializována hodnotou $false$ a určuje, zda se dotaz zadává v rámci stejného bloku nebo nikoliv. Proměnná *typ* určuje, zda příští dotaz je unární (když $typ = true$) nebo binární. Proměnná *smer* určuje, zda dotazy jsou menší než první dotaz v bloku ($smer = true$) nebo větší. Dále procedura používá proměnnou *krok* typu integer, která obsahuje velikost kroku v rámci bloku. Hodnoty těchto proměnných se předávají z jednoho volání procedury do dalšího volání (tj. jsou to globální proměnné, které se neinicializují voláním procedury **next**).

```
  next(d, h)
if blok then
  if typ then
    if smer then
      next(d, h) := h − krok
      if A (next (d, h)) < x then
        blok := false
      endif
    else
      next(d, h) := d + krok
      if A (next (d, h)) > x then
        blok := false
      endif
    endif
    typ := false
  else
    next(d, h) := ⌈(d+h)/2⌉
    if A (next (d, h)) > x a (d−h)/2 < krok then
      blok := false
    else
      typ := true
    endif
  endif
else
  krok := ⌊√(h − d)⌋, next(d, h) := d + ⌈((x−A(d))/(A(h)−A(d))) (h − d)⌉,
  if A (next (d, h)) > x then
    smer := true
  else
    smer := false
  endif
  typ := true, blok := true
endif
```

Po dvou dotazech klesne $h-d$ buď pod $\sqrt{h-d}$ nebo pod $\frac{h+d}{2}$. Proto procedura v nejhorším případě použije nejvýše $8+2\log\left(|S|-1\right)+2\log\log|S|$ dotazů, a tedy v nejhorším případě vyžaduje čas $O\left(\log|S|\right)$.

Nyní spočítáme očekávaný počet dotazů během jednoho bloku za předpokladu rovnoměrného rozdělení dat. Nechť p_i je pravděpodobnost, že v rámci bloku se položí alespoň i dotazů. Pak očekávaný počet dotazů v rámci bloku je

$$E\left(C\right)=\sum_{i\geq 1} i\left(p_i-p_{i+1}\right)=\sum_{i\geq 1} p_i.$$

Nyní odhadneme p_i. Označme $n+d$ argument prvního dotazu (interpolační vyhledávaní) v rámci bloku a nechť $krok=k$ v rámci bloku. Označme $X=\left|\,\{i\mid i>d,\,A\left(i\right)\leq x\}\,\right|$ na začátku bloku, pak X je náhodná proměnná závislá na argumentu operace a bloku. Když se v bloku položí alespoň i dotazů pro $i>2$, pak $|X-n|\geq\lfloor\frac{i-2}{2}\rfloor k$, protože každý unární dotaz, jehož položení nezmění blok, nalezne dalších k hodnot i v rozdílu $|X-n|$. Tedy

$$p_i \leq \text{Prob}\left(|X-n| \geq \lfloor\frac{i-2}{2}\rfloor k\right).$$

Použijeme Čebyševovu nerovnost pro náhodnou proměnnou X. Když Y je náhodná proměnná s očekávanou (střední) hodnotou μ a rozptylem σ^2, pak Čebyševova nerovnost říká, že

$$\text{Prob}\,(|Y-\mu| \geq t) \leq \frac{\sigma^2}{t^2} \qquad \text{pro každé } t > 0.$$

Uvažujme okamžik, kdy jsme na začátku nějakého bloku. Protože S je vybraná s rovnoměrným rozdělením, je pravděpodobnost, že $A\,(i) < x$ pro $d < i < h$, rovna $p = \frac{x-A(d)}{A(h)-A(d)}$, a pak pravděpodobnost, že $X = j$, je $\binom{h-d}{j} p^j\,(1-p)^{h-d-j}$. To znamená, že X je náhodná veličina s binomickým rozdělením s rozsahem $d-h$ a pravděpodobností p, a tedy její očekávaná hodnota je

$$\mu = \sum_{j=0}^{h-d} j \binom{h-d}{j} p^j\,(1-p)^{h-d-j} = p\,(h-d)$$

a rozptyl má hodnotu

$$\sigma^2 = \sum_{j=0}^{h-d} (j-\mu)^2 \binom{h-d}{j} p^j\,(1-p)^{h-d-j} = p\,(1-p)\,(h-d)\,.$$

Když si uvědomíme, že $k = \lfloor\sqrt{h-d}\rfloor$ a $n = p\,(h-d)$, pak dostáváme

$$p_i, p_{i+1} \leq \text{Prob}\left(|X-n| \geq \lfloor\frac{i-2}{2}\rfloor k\right) \leq \frac{4p\,(1-p)\,(h-d)}{(i-2)^2\,k^2} \leq$$
$$\frac{4p\,(1-p)}{(i-2)^2} \leq \frac{1}{(i-2)^2},$$

protože pro $0 \leq p \leq 1$ je $p\,(1-p) \leq \frac{1}{4}$. Když shrneme tato pozorování, dostáváme, že

$$E\,(C) = \sum_{i\geq 1} p_i \leq 2 + 2\sum_{i\geq 3} \frac{1}{(i-2)^2} = 2 + 2\sum_{i\geq 1}\frac{1}{i^2} =$$
$$2 + 2\frac{\pi^2}{6} = 2 + \frac{\pi^2}{3} \approx 5.3$$

Závěr: očekávaný počet dotazů v bloku je menší než 6.

Když $E\,(T\,(n))$ je očekávaný počet dotazů pro operaci **MEMBER** a když $|S| = n$, pak platí

$$E\,(T\,(n)) \leq E\,(C) + E\left(T\left(\sqrt{n}\right)\right).$$

Protože $E\,(T\,(1)) = 1$ a $E\,(T\,(2)) \leq 2$, dostáváme z rekurentního vzorce, že

$$E\,(T\,(n)) \leq 2 + E\,(C) \log\log n \qquad \text{pro } n \geq 2.$$

Věta. *Čas operace* **MEMBER** *v uspořádaném poli délky n při zobecněném kvadratickém vyhledávání je v nejhorším případě $O\,(\log n)$. Když rozdělení vstupních dat je rovnoměrné, pak očekávaný čas je $O\,(\log\log n)$.*

Nevýhoda této datové struktury spočívá v neexistenci přirozených efektivních implementací operací **INSERT**, **DELETE**, **SPLIT** a **JOIN**. Přirozené implementace těchto operací vyžadují čas $O\,(|S|)$, zhruba řečeno musíme pohybovat s téměř každým prvkem. Pokusem o řešení tohoto problému byl návrh binárních vyhledávacích stromů.

Binární vyhledávací stromy

Binární vyhledávací strom je struktura pro binární vyhledávání v uspořádaném poli roztaženém do roviny a vyhledávání odpovídá cestě ve stromě. Formální definice:
Předpokládáme, že U je lineárně uspořádané univerzum a $S \subseteq U$. Binární vyhledávací strom T reprezentující množinu S je úplný binární strom (tj. každý vrchol je buď listem nebo má dva syny, levého a pravého), kde existuje bijekce mezi množinou S a vnitřními vrcholy stromu taková, že

> když v je vnitřní vrchol stromu T, kterému je přiřazen prvek $s \in S$, pak každému vnitřnímu vrcholu u v podstromu levého syna vrcholu v je přiřazen prvek z S menší než s a každému vnitřnímu vrcholu w v podstromu pravého syna vrcholu v je přiřazen prvek z S větší než s.

Strukura vnitřního vrcholu v:
ukazatel $\text{otec}(v)$ na otce vrcholu v,
ukazatel $\text{levy}(v)$ na levého syna vrcholu v,
ukazatel $\text{pravy}(v)$ na pravého syna vrcholu v,
atribut $\text{key}(v)$ – prvek z S přiřazený vrcholu v. Když v je kořen stromu, pak hodnota ukazatele $\text{otec}(v)$ je NIL. List má ukazatele pouze na otce.

Každý list reprezentuje interval mezi dvěma sousedními prvky z S – přesně, když u je list a je levým synem vrcholu v, nalezneme vrchol na cestě z u do kořene nejblíže u takový, že je pravým synem vrcholu w. Pak u reprezentuje interval $(\text{key}(w), \text{key}(v))$ a když vrchol w neexistuje, pak u reprezentuje interval $(-\infty, \text{key}(v))$ a prvek $\text{key}(v)$ je nejmenší prvek v S. Když u je list a je pravým synem vrcholu v, nalezneme vrchol na cestě z u do kořene nejblíže u takový, že je levým synem vrcholu w. Pak u reprezentuje interval $(\text{key}(v), \text{key}(w))$ a když takový vrchol w neexistuje, pak u reprezentuje interval $(\text{key}(v), +\infty)$ a prvek $\text{key}(v)$ je největší prvek v S.

Při implementaci binárních vyhledávácích stromů je výhodné vynechat listy (místo nich bude ukazatel NIL). Při návrhu algoritmů je však naopak výhodné pracovat s listy (vyhlíží to logičtější). Proto při návrhu algoritmů budeme předpokládat, že stromy mají listy reprezentující intervaly.

Navrhneme algoritmy pro binární vyhledávací stromy realizující operace z uspořádaného slovníkového problému.

Vyhledej(x)
t :=kořen stromu
while t není list a $\text{key}(t) \neq x$ **do**
  if $\text{key}(t) > x$ **then** $t := \text{levy}(t)$ **else** $t := \text{pravy}(t)$ **endif**
enddo

MEMBER(x)
Vyhledej(x)
if t není list **then Výstup**: $x \in S$ **else Výstup**: $x \notin S$ **endif**

INSERT(x)
Vyhledej(x)
if t je list **then**
  t se změní na vnitřní vrchol, $\text{key}(t) := x$,
  $\text{levy}(t)$ a $\text{pravy}(t)$ jsou nové listy, jejichž otcem je t
endif

DELETE(x)
```
Vyhledej(x)
if t není list then
   if levy (t) je list then
      odstraníme vrchol levy (t), otec (pravy (t)) := otec (t)
      if t = levy (otec (t)) then
         levy (otec (t)) := pravy (t)
      else
         pravy (otec (t)) := pravy (t)
      endif
      odstraníme vrchol t
   else
      u := levy (t)
      while pravy (u) není list do
         u := pravy (u)
      enddo
      key (t) := key (u), odstraníme vrchol pravy (u),
      otec (levy (u)) := otec (u)
      if u = levy (otec (u)) then
         levy (otec (u)) := levy (u)
      else
         pravy (otec (u)) := levy (u)
      endif
      odstraníme vrchol u
   endif
endif
```

MIN
```
t :=kořen stromu
while levý syn t není list do t := levy (t) enddo
Výstup: prvek reprezentovaný t je nejmenší prvek v S
```

MAX
```
t :=kořen stromu
while pravý syn t není list do t := pravy (t) enddo
Výstup: prvek reprezentovaný t je největší prvek v S
```

SPLIT(x):
```
T1 a T2 jsou prázdné stromy
u1 := u2 := NIL
t := kořen stromu T
while t není list a key (t) ≠ x do
   if key (t) > x then
      u := levy (t), levy (t) := NIL, otec (u) := NIL
      if T2 je prázdný strom then
         T2 := podstrom vrcholu t
      else
         levy (u2) := t, otec (t) := u2
      endif
      u2 := t
```

else
$u := \text{pravy}\,(t)$, $\text{pravy}\,(t) := NIL$, $\text{otec}\,(u) := NIL$
if T_1 je prázdný strom **then**
$T_1 :=$ podstrom vrcholu t
else
$\text{pravy}\,(u_1) := t$, $\text{otec}\,(t) := u_1$
endif
$u_1 := t$
endif
$t := u$
enddo
if $\text{key}\,(t) = x$ **then**
$\text{otec}\,(\text{levy}\,(t)) := u_1$, $\text{pravy}\,(u_1) := \text{levy}\,(t)$
$\text{otec}\,(\text{pravy}\,(t)) := u_2$, $\text{levy}\,(u_2) := \text{pravy}\,(t)$
$\text{otec}\,(u_1) := NIL$, $\text{otec}\,(u_2) := NIL$, **Výstup**: $x \in S$
else
Výstup: $x \notin S$
endif
Komentář: T_1 je binární vyhledávací strom reprezentující množinu $\{s \in S \mid s < x\}$ a T_2 je binární vyhledávací strom reprezentující množinu $\{s \in S \mid s > x\}$.

JOIN3(T_1, x, T_2) – předpokládáme, že když T_i reprezentuje množinu S_i pro $i = 1, 2$, pak $\max S_1 < x < \min S_2$
vytvořme nový vrchol u, $\text{key}\,(u) = x$, $\text{otec}\,(u) := NIL$,
otec (kořene T_1) $:= u$, otec (kořene T_2) $:= u$,
$\text{levy}\,(u) :=$kořen T_1, $\text{pravy}\,(u) :=$kořen T_2.

Abych dokázali korektnost algoritmu **Vyhledej** – jedná se o modifikaci vyhledávání v uspořádaném poli – popíšeme podrobněji vlastnosti binárního vyhledávacího stromu. Nejprve rozšíříme universum o dva nové prvky, o nový nejmenší prvek $-\infty$ a o nový největší prvek $+\infty$. Mějme binární vyhledávací strom T reprezentující množinu S, pak pro vrchol t stromu T definujeme indukcí hodnoty $\lambda\,(t)$ a $\pi\,(t)$. Když r je kořen, pak $\lambda\,(r) = -\infty$ a $\pi\,(r) = +\infty$. Když hodnoty $\lambda\,(t)$ a $\pi\,(t)$ jsou pro vrchol t definovány, pak pro levého syna u vrcholu t definujeme $\lambda\,(u) = \lambda\,(t)$ a $\pi\,(u) = \text{key}\,(y)$ a pro pravého syna w vrcholu t definujeme $\lambda\,(w) = \text{key}\,(t)$ a $\pi\,(w) = \pi\,(t)$. Nyní dokážeme

Lemma. *Je-li T' podstrom binárního vyhledávacího stromu T určený vrcholem t, pak T' reprezentuje množinu $S \cap (\lambda\,(t)\,, \pi\,(t))$. Navíc interval $(\lambda\,(t)\,, \pi\,(t))$ je největší interval, který obsahuje jenom prvky z S, které jsou reprezentovány vrcholy podstromu T'. Navíc, když t je list, pak $< \lambda\,(t)\,, \pi\,(t) >$ je interval reprezentovaný listem t.*

Důkaz. Tvrzení dokážeme indukcí. Zřejmě platí, když t je kořen stromu T. Předpokládejme, že platí pro vrchol t a dokážeme ho pro syny vrcholu t. Označme t_l levého syna vrcholu t, t_p pravého syna vrcholu t. Z definice binárního vyhledávacího stromu stromu plyne, že když u je vnitřní vrchol v podstromu T určeném vrcholem t_l a když v je vnitřní vrchol v podstromu T určeném vrcholem t_p, pak $\text{key}\,(u) < \text{key}\,(t) < \text{key}\,(v)$. Nyní platnost tvrzení pro t implikuje platnost tvrzení i pro vrcholy t_l a t_p. $\square$

Korektnost podprocedury **Vyhledej** plyne z následujícího invariantu:

Když při vyhledávání x vyšetřujeme vrchol t, pak

$$\lambda\,(t) < x < \pi\,(t)\,.$$

Toto tvrzení se lehce dokáže indukcí z popisu algoritmu **Vyhledej**. Tedy operace **Vyhledej** je korektní a korektnost operací **MEMBER** a **INSERT** je teď zřejmá. V operaci **DELETE**, když levy (t) je list, pak korektnost je zřejmá. Když levy (t) není list, pak algoritmus nalezne list v takový, že $\pi(v) = x$. Pak pro $u = \text{otec}(v)$ platí $v = \text{pravy}(u)$ a $\lambda(v) = \text{key}(u)$ a $(\lambda(v), \pi(v)) \cap S = \emptyset$. Když $y = \text{key}(u)$, pak odstranění vrcholů u a v dává binární vyhledávací strom reprezentující $S \setminus \{y\}$. Protože $(y, x) \cap S = \emptyset$, tak příkaz key $(t) := y$ dává binární vyhledávací strom reprezentující $S \setminus \{x\}$ a proto operace **DELETE** je korektní.

Korektnost operací **MIN**, **MAX** a **JOIN3** plyne z definice binárního vyhledavacího stromu. Korektnost operace **SPLIT** plyne z korektnosti algoritmu **Vyhledej** a z faktu, že u_1 je otec nejpravějšího listu stromu T_1 a u_2 je otec nejlevějšího listu stromu T_2. Protože ke stromu T_1 se přidává část stromu T reprezentující prvky, které jsou větší než prvky reprezentované v T_1, a ke stromu T_2 se přidává část stromu T reprezentující prvky, které jsou menší než prvky reprezentované v T_2, korektnost algoritmu pro operaci **SPLIT** je jasná.

Zpracování jednoho vrcholu vyžaduje čas $O(1)$ a algoritmus se pohybuje po jedné cestě z kořene do nějakého listu. Označme hloubka (T) délku nejdelší cesty z kořene do nějakého listu. Pak dostáváme

Věta. *Algoritmy pro operace* **MEMBER**, **INSERT**, **DELETE**, **MIN**, **MAX**, **JOIN3** *a* **SPLIT** *v binárním vyhledávacím stromě T vyžadují čas $O(\text{hloubka}(T))$.*

Bohužel ani struktura binárních vyhledávacích stromů nepodporuje efektivní implementaci operace **ord**(k). Pro její efektivní implentaci je vhodné rozšířit datovou strukturu tak, že u každého vrcholu t je deklarován také údaj $p(t)$ – počet listů v podstromu určeném vrcholem t. Po provedení operací **INSERT**, **DELETE**, **JOIN3** a **SPLIT** je pak nutné aktualizovat tuto položku na cestě z vrcholu do kořene. Následující algoritmus pak realizuje operaci **ord**(k).

ord(k)
t :=kořen stromu
if $k \geq p(t)$ **then** k-tý prvek neexistuje, **stop endif**
while true **do**
  if $k > p(\text{levy}(t))$ **then**
    $k := k - p(\text{levy}(t))$, $t := \text{pravy}(t)$
  else
    if $k < p(\text{levy}(t))$ **then**
      $t := \text{levy}(t)$
    else
      key (t) je k-tý prvek reprezentované množiny, stop
    endif
  endif
enddo

Korektnost algoritmu plyne z následujícího invariantu: Když algoritmus má v daném okamžiku v proměnné t vrchol v a hodnota proměnné k je k', pak k-tý prvek v S se rovná k'-tému prvku v intervalu reprezentovaném v podstromu stromu T určeném vrcholem v. Protože na počátku algoritmu je v kořen stromu a interval je S (a $k' = k$), tak na počátku běhu algoritmu invariant platí. Předpokládejme, že platí v některém kroku. Nechť u

je levý syn v, w je pravý syn v a I_a je interval reprezentovaný podstromem T určeným vrcholem a. Pak $|I_u| = p(u) - 1$, $\max I_u < \text{key}(v) < \min I_w$ a $I_v = I_u \cup \{\text{key}(v)\} \cup I_w$. Odtud plyne, že když $k' < p(u)$, pak k'-tý prvek v intervalu I_v je k'-tý prvek v intervalu I_u, když $k' > p(u)$, pak k'-tý prvek v intervalu I_v je $(k' - p(u))$-tý prvek v intervalu I_w, a když $k' = p(u)$, pak k'-tý prvek v intervalu I_v je $\text{key}(v)$. Odtud plyne invariant a korektnost algoritmu. Podle stejných argumentů jako v předchozím případě dostaneme, že časová složitost algoritmu je $O(\text{hloubka}(T))$. Tedy můžeme tato fakta shrnout.

Věta. *Algoritmy pro operace* **MEMBER***,* **INSERT***,* **DELETE***,* **MIN***,* **MAX***,* **JOIN3***,* **SPLIT** *a* **ord** (k) *pro všechna* k *v rozšířených binárních vyhledávacích stromech vyžadují čas* $O(\text{hloubka}(T))$*, kde* T *je strom reprezentující danou množinu.*

Tento výsledek motivuje používání binárních vyhledávacích stromů, které splňují další podmínku, která má zajistit, že $\text{hloubka}(T) = O(\log |S|)$. V takovémto případě mluvíme o <u>vyvážených</u> <u>binárních</u> <u>vyhledávacích</u> <u>stromech</u>. Je však nutné přidat k operacím **INSERT**, **DELETE**, **JOIN3** a **SPLIT** další kroky, které zaručí, že po jejich provedení strom opět splňuje požadované podmínky. To vede k požadavku, aby vyvažovací operace byly rychlé a provádělo se jich málo.

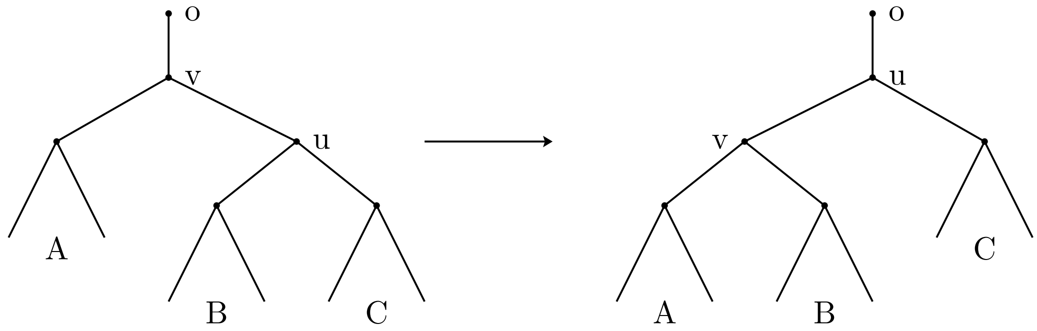
Při náhodné posloupnosti operací **INSERT** a **DELETE** je velká pravděpodobnost, že dostaneme náhodný binární vyhledávací strom. Je známo, že očekávaná hodnota proměnné $\text{hloubka}(T)$ je $O(\log |S|)$. Protože se nepoužívají vyvažovací operace, můžeme dostat lepší výsledek (časově) než pro vyvážené binární vyhledávací stromy. Tento problém se teď intenzivně studuje. Velká pozornost je věnována pravděpodobnostním modifikacím binárních vyhledávacích stromů. Hledají se však i další možnosti.

Studují se tzv. samoupravující struktury. Zde se pracuje s datovou strukturou bez dodatečných informací, ale operace nad touto strukturou provádí vyvažování v závislosti na argumentu operace. Dokázalo se, že existuje strategie vyvažování, která zajišťuje dobré chování bez ohledu na vstupní data. Další strategie je, že se jen zjišťuje, zda datová struktura nemá výrazně špatné chování, a pokud ho má nebo po dlouhé řadě úspěšných aktualizačních operací se vybuduje nová datová struktura (s optimálním chováním). Třetí, poměrně stará, strategie je založena na předpokladu, že známe rozdělení vstupních dat. Zde se datová struktura předem upravuje pro toto rozdělení. Ukazuje se, že tyto strategie mají úspěch. Další podrobnosti v letním semestru.

Nyní si ukážme dvě operace se stromy, na nichž jsou založeny vyvažovací operace pro binární vyhledávací stromy. Obě operace vyžadují čas $O(1)$.

Mějme vrchol v binárního vyhledávacího stromu T a jeho syna u, který je vnitřní vrchol. Pak **Rotace**(v, u) je znázorněna na obrázku a provádí ji následující algoritmus.

Rotace(v, u)
otec (u) := otec (v),
if v = levy (otec (v)) **then**
levy (otec (v)) := u
else
pravy (otec (v)) := u
endif
otec (v) := u
if u = levy (v) **then**

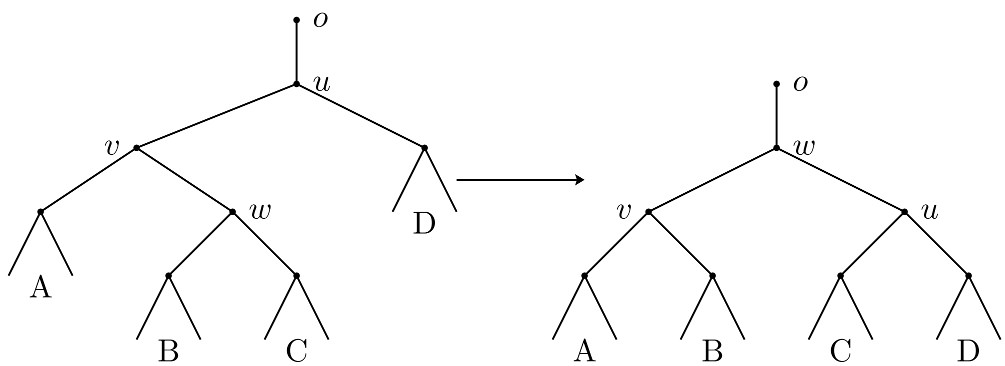


Obr. 1

otec (pravy (u)) := v, levy (v) := pravy (u), pravy (u) := v
else
otec (levy (u)) := v, pravy (v) := levy (u), levy (u) := v
endif

Všimněme si, že při **Rotace** můžeme aktualizovat i funkci p. Pro vrchol $w \neq u, v$ se její hodnota nemění, nová hodnota $p(u)$ je rovná původní hodnotě $p(v)$ a novou hodnotu $p(v)$ dostaneme jako $p(\text{levy}(v)) + p(\text{pravy}(v))$.

Mějme vrchol u stromu T, jeho syna v a jeho syna w takového, že w není list a v je pravý syn vrcholu u, právě když w je levý syn vrcholu v. Pak **Dvojita-rotace**(u, v, w) je znázorněna na obrázku a provádí ji následující algoritmus.



Obr. 2

Dvojita-rotace(u, v, w)
otec (w) := otec (u)
if u = levy (otec (u)) **then**
levy (otec (w)) := w
else
pravy (otec (w)) := w
endif
otec (v) := w, otec (u) := w
if v = levy (u) **then**

levy (u) := pravy (w), otec (pravy (w)) := u, pravy (v) := levy (w)
otec (levy (w)) := v, levy (w) := v, pravy (w) := u
else
pravy (u) := levy (w), otec (levy (w)) := u, levy (v) := pravy (w)
otec (pravy (w)) := v, levy (w) := u, pravy (w) := v
endif

Také zde můžeme v čase $O(1)$ spočítat nové hodnoty p. Pro vrchol $x \neq u, v, w$ se hodnota nemění, nová hodnota $p(w)$ je rovná původní hodnotě $p(u)$ a nové hodnoty $p(u)$ a $p(v)$ získáme podle stejného vzorce jako v **Rotace**.

AVL-stromy

Binární vyhledávací strom je AVL-strom, když pro každý vnitřní vrchol v se délka nejdelší cesty z jeho levého syna do listu a délka nejdelší cesty z jeho pravého syna do listu liší nejvýše o 1.

Pro vnitřní vrchol v stromu T označme $\eta(v)$ délku nejdelší cesty z vrcholu v do listu.

Struktura vnitřních vrcholů v AVL-stromech je rozšířena o hodnotu ω:
$\omega(v) = -1$, když

$$\eta(\text{levý syn vrcholu } v) = \eta(\text{pravý syn vrcholu } v) + 1;$$

$\omega(v) = 0$, když

$$\eta(\text{levý syn vrcholu } v) = \eta(\text{pravý syn vrcholu } v);$$

$\omega(v) = +1$, když

$$\eta(\text{levý syn vrcholu } v) + 1 = \eta(\text{pravý syn vrcholu } v).$$

Všimněme si, že hodnota $\eta(v)$ pro vnitřní vrcholy v stromu T není nikde uložena. Hodnoty η jsme schopni spočítat z hodnot ω, ale není to třeba. Stačí, když po aktualizačních operacích budeme umět aktualizovat hodnoty ω a upravit binární vyhledávací strom tak, aby byl opět AVL-strom.

Odhad velikosti $\eta(\text{kořen } T)$ v závislosti na velikosti reprezentované množiny S.
Když T je AVL-strom a v je vnitřní vrchol T, pak podstrom T určený vrcholem v je opět AVL-strom. Označme
$mn(i)$ velikost nejmenší množiny reprezentované AVL-stromem T takovým, že

$$\eta(\text{kořen } T) = i,$$

$mx(i)$ velikost největší množiny reprezentované AVL-stromem T takovým, že

$$\eta(\text{kořen } T) = i.$$

Z definice AVL-stromu plynou rekurze

$$mn(i) = mn(i-1) + mn(i-2) + 1,\ mx(i) = 2mx(i-1) + 1,$$
$$\text{a } mn(1) = mx(1) = 1, \quad mn(2) = 2, \quad mx(2) = 3.$$

Nejprve spočítáme mx.

Dokážeme, že $mx(i) = 2^i - 1$. Tento vzorec je splněn pro $i = 1, 2$. Dále

$$mx(i+1) = 2mx(i) + 1 = 2\left(2^i - 1\right) + 1 = 2^{i+1} - 1.$$

Tím je vzorec dokázán.

Abychom spočítali mn, připomeneme si definici Fibonacciho čísel. Fibonacciho číslo F_i je definováno rekurencí

$$F_1 = F_2 = 1 \text{ a } F_{i+2} = F_i + F_{i+1} \text{ pro všechna } i \geq 3.$$

Pak platí vzorec $F_i = \frac{\left(\frac{1+\sqrt{5}}{2}\right)^i - \left(\frac{1-\sqrt{5}}{2}\right)^i}{\sqrt{5}}$ pro všechna $i \geq 1$ (dokážeme si ho v části o haldách).

Protože $-1 < \frac{1-\sqrt{5}}{2} < 0$ a $\frac{1+\sqrt{5}}{2} > 1$, dostáváme, že

$$\lim_{n \mapsto \infty} F_n \sqrt{5} \left(\frac{1+\sqrt{5}}{2}\right)^{-n} = 1.$$

Proto existují konstanty $0 < c_1 < c_2$ takové, že

$$c_1 \left(\frac{1+\sqrt{5}}{2}\right)^i < \sqrt{5} F_i < c_2 \left(\frac{1+\sqrt{5}}{2}\right)^i.$$

Dokážeme, že $mn(i) = F_{i+2} - 1$. Protože $F_3 = 2$ a $F_4 = 3$, tvrzení platí pro $i = 1$ a $i = 2$. Dále

$$\begin{aligned} mn(i+2) =& mn(i+1) + mn(i) + 1 = \\ & F_{i+3} - 1 + F_{i+2} - 1 + 1 = F_{i+4} - 1. \end{aligned}$$

Z toho indukcí plyne požadovaný vztah.

Když AVL-strom T o výšce i reprezentuje množinu S o velikosti n, pak platí

$$\frac{c_1}{\sqrt{5}} \left(\frac{1+\sqrt{5}}{2}\right)^{i+2} - 1 < F_{i+2} - 1 \leq n \leq 2^i - 1.$$

Po zlogaritmování z toho okamžitě dostáváme

$$\log\left(\frac{c_1}{\sqrt{5}}\right) + (i+2) \log\left(\frac{1+\sqrt{5}}{2}\right) < \log(n+1) < i.$$

Protože $\log\left(\frac{1+\sqrt{5}}{2}\right) \approx 0.69 \approx \frac{1}{1.44}$ dostáváme, že pro dostatečně velká n platí, že $0.69i < \log(n+1) \leq i$. Odtud plyne, že $\log(n+1) \leq i \leq 1.44 \log(n)$, a tedy $i = \Theta(\log(n))$.

Operace **MEMBER**(x) pro AVL-stromy je stejná jako operace **MEMBER**(x) pro nevyvážené binární vyhledávací stromy. Aktualizační operace pro AVL-stromy nejprve provedou příslušnou operaci pro nevyvážené binární vyhledávací stromy a pak následuje jejich vyvažovací část. Při úspěšně provedené operaci **INSERT**(x) v nevyvážených binárních stromech změníme vhodný list t na vnitřní vrchol stromu reprezentující x a přidáme k t dva syny, kteří budou listy. Důsledkem je, že definujeme $\omega(t) = 0$. Protože se však zvětšila hodnota $\eta(t)$ (bylo $\eta(t) = 0$ a teď je $\eta(t) = 1$), zavoláme proceduru **Kontrola-INSERT**(t), která zajistí správnou hodnotu funkce ω pro otce t. Navíc, když zjistí, že se zvětšila hodnota η vrcholu otce t, pak zavolá sama sebe na vrchol otec t. Nejprve provedeme analýzu situace.

Mějme vrchol t, jeho $\eta(t) = a$ (ale a neznáme), na začátku operace **INSERT** bylo $\eta(t) = a - 1$. V podstromu určeném vrcholem t máme už správné hodnoty ω. Vrchol v je otcem t, $t = \text{levy}(v)$ a $\omega(v)$ má ještě původní hodnotu.

Lemma. *Když se hodnota $\eta(t)$ při operaci* **INSERT** *zvětšila a t nebyl listem před operací, pak po operaci neplatí $\omega(t) = 0$.*

Označme $u = \text{bratr}(t) = \text{pravy}(v)$ a uvažme případy:
A) $\omega(v) = 1$, pak $\eta(u) = a$ a $\eta(v) = a + 1$ se nezměnilo, tedy stačí položit $\omega(v) = 0$.
B) $\omega(v) = 0$, pak $\eta(u) = a - 1$ a $\eta(v) = a + 1$ se změnilo, tedy musíme položit $\omega(v) = -1$ a zavolat proceduru **Kontrola-INSERT** na vrchol v.
C) $\omega(v) = -1$, pak $\eta(u) = a - 2$ a $\eta(v) = a + 1$ se změnilo. Nyní $\omega(v) = -2$ a to je zakázané. Označme $t_1 = \text{levy}(t)$, $t_2 = \text{pravy}(t)$ ($\omega(t) = 0$ nenastane, viz Lemma).
C1) $\omega(t) = -1$, pak $\eta(t_1) = a - 1$, $\eta(t_2) = a - 2$ a provedeme **Rotace**(v, t). Pak t_2 je druhý syn v a stačí položit $\omega(v) = \omega(t) = 0$.
C2) $\omega(t) = 1$, pak $\eta(t_1) = a - 2$, $\eta(t_2) = a - 1$ a provedeme **Dvojita-rotace**(v, t, t_2). Pro $t_3 = \text{levy}(t_2)$ a $t_4 = \text{pravy}(t_2)$ platí:
C2i) $\omega(t_2) = 1 \implies \eta(t_3) = a - 3$ a $\eta(t_4) = a - 2$ a stačí položit $\omega(t) = -1$, $\omega(v) = \omega(t_2) = 0$, protože $\eta(t_2) = a$.
C2ii) $\omega(t_2) = 0 \implies \eta(t_3) = \eta(t_4) = a - 2$ a stačí položit $\omega(t_2) = \omega(v) = \omega(t) = 0$, protože $\eta(t_2) = a$.
C2iii) $\omega(t_2) = -1 \implies \eta(t_3) = a - 2$ a $\eta(t_4) = a - 3$ a stačí položit $\omega(v) = 1$, $\omega(t_2) = \omega(t) = 0$, protože $\eta(t_2) = a$.
Když t je pravý syn v, pak situace je symetrická.
Popíšeme proceduru **Kontrola-INSERT**.

Kontrola-INSERT(t)
$v :=$ otec(t)
if $t =$ levy(v) **then**
  if $\omega(v) = 1$ **then**
    $\omega(v) := 0$
  else
    if $\omega(v) = 0$ **then**
      $\omega(v) := -1$, $t := v$, **Kontrola-INSERT**(t)
    else
      if $\omega(t) = -1$ **then**
        Rotace(v, t), $\omega(v) := 0$, $\omega(t) := 0$
      else
        $t_2 :=$ pravy(t), **Dvojita-rotace**(v, t, t_2),
        if $\omega(t_2) = 0$ **then**

```
            ω(t) := 0, ω(v) := 0
          else
            if ω(t_2) = 1 then
              ω(v) := 0, ω(t) := −1
            else
              ω(v) := 1, ω(t) := 0
            endif
          endif
          ω(t_2) := 0
        endif
      endif
    endif
else
  if ω(v) = −1 then
    ω(v) := 0
  else
    if ω(v) = 0 then
      ω(v) := 1, t := v, Kontrola-INSERT(t)
    else
      if ω(t) = 1 then
        Rotace(v, t), ω(v) := 0, ω(t) := 0
      else
        t_1 := levy(t), Dvojita-rotace(v, t, t_1),
        if ω(t_1) = 0 then
          ω(t) := 0, ω(v) := 0
        else
          if ω(t_1) = 1 then
            ω(v) := 0, ω(t) := −1
          else
            ω(v) := 1, ω(t) := 0
          endif
        endif
        ω(t_1) := 0
      endif
    endif
  endif
endif
```

Všimněme si, že po provedení **Rotace** nebo **Dvojita-rotace** vyvažování v operaci **INSERT** končí. Tedy operace **INSERT** provádí nejvýše jednu proceduru **Rotace** nebo **Dvojita-rotace**. Korektnost vyvažovací operace je založena na faktu, že když se zvětší hodnota $\eta(t)$, pak nemůže být $\omega(t) = 0$. Tento fakt se využívá v **if**-příkazu na 5-tém a 6-tém řádku a na 13-tém a 14-tém řádku programu.

Popíšeme vyvažovací operaci pro operaci **DELETE**. Předpokládejme, že t je vrchol, jehož otec se odstranil (tj. bratr t byl list) a hodnota $\eta(t)$ je menší než byla hodnota $\eta(\text{otec}(t))$. Proto zavoláme proceduru **Kontrola-DELETE**(t). Tato procedura zajistí správnou hodnotu funkce ω pro otce t. Navíc, když zjistí, že se zmenšila hodnota η vrcholu otce t, pak zavolá sama sebe na vrchol otec t. Popíšeme analýzu situace, na níž je založena korektnost procedury **Kontrola-DELETE**(t).

V analýze je důležité, že když procedura **Kontrola-DELETE** přesune vrchol x na místo vrcholu y, pak skutečná hodnota $\eta(x)$ je buď původní hodnota $\eta(y)$ nebo je přesně o 1 menší. Všimněte si, že to platí.

Dán vrchol t, jehož hodnota $\eta(t)$ se zmenšila (o 1). V podstromu určeném vrcholem t jsou hodnoty ω aktualizovány, $v = \text{otec}(t)$ a $\omega(v)$ je původní. Předpokládejme $t = \text{levy}(v)$, $u = \text{bratr}(t) = \text{pravy}(v)$ a $\eta(t) = a$ (a je neznámé). Nastávají případy:
A) když $\omega(v) = 1$, pak $\eta(u) = a+2$ a $\eta(v) = a+3$ (původně bylo $\eta(t) = a+1$). Označme $u_1 = \text{levy}(u)$, $u_2 = \text{pravy}(u)$.
A1) $\omega(u) = 1 \implies \eta(u_1) = a, \eta(u_2) = a+1$. Provedeme **Rotace**$(v, u)$. Vrchol u_1 je druhým synem v a platí $\eta(t) = \eta(u_1) = a$, $\eta(v) = \eta(u_2) = a+1$ a $\eta(u) = a+2$. Tedy položme $\omega(v) = \omega(u) = 0$ a zavolejme **Kontrola-DELETE** na vrchol u.
A2) $\omega(u) = 0 \implies \eta(u_1) = \eta(u_2) = a+1$. Provedeme **Rotace**$(v, u)$. Vrchol u_1 je druhým synem v a platí $\eta(t) = a$, $\eta(u_1) = a+1 = \eta(u_2)$, $\eta(v) = a+2$, $\eta(u) = a+3$. Položme $\omega(v) = 1$, $\omega(u) = -1$ a končíme.
A3) $\omega(u) = -1 \implies \eta(u_1) = a+1, \eta(u_2) = a$. Provedeme **Dvojita-rotace**(v, u, u_1). Pro $u_3 = \text{levy}(u_1)$, $u_4 = \text{pravy}(u_1)$ nastanou případy:
A3i) $\omega(u_1) = -1 \implies \eta(u_3) = a, \eta(u_4) = a-1$ a tedy $\eta(v) = \eta(u) = a+1$ a $\eta(u_1) = a+2$. Proto položíme $\omega(v) = \omega(u_1) = 0$, $\omega(u) = 1$ a zavoláme proceduru **Kontrola-DELETE** na vrchol u_1.
A3ii) $\omega(u_1) = 0 \implies \eta(u_3) = \eta(u_4) = a$ a tedy $\eta(v) = \eta(u) = a+1$ a $\eta(u_1) = a+2$. Proto položíme $\omega(v) = \omega(u_1) = \omega(u) = 0$ a zavoláme proceduru **Kontrola-DELETE** na vrchol u_1.
A3iii) $\omega(u_1) = 1 \implies \eta(u_3) = a-1, \eta(u_4) = a$ a tedy $\eta(v) = \eta(u) = a+1$ a $\eta(u_1) = a+2$. Proto položíme $\omega(u) = \omega(u_1) = 0$, $\omega(v) = -1$ a zavoláme proceduru **Kontrola-DELETE** na vrchol u_1.
B) když $\omega(v) = 0$, pak $\eta(u) = a+1$ a $\eta(v) = a+2$. Stačí položit $\omega(v) = 1$ a skončit.
C) když $\omega(v) = -1$, pak $\eta(u) = a$ a $\eta(v) = a+2$. Nyní položíme $\omega(v) = 0$ a zavoláme proceduru **Kontrola-DELETE** na vrchol v.

```
  Kontrola-DELETE(t)
v := otec(t), u := bratr(t)
if t = levy(v) then
  if ω(v) = 1 then
    if ω(u) ≥ 0 then
      Rotace(v, u)
      if ω(v) = 0 then
        ω(v) := 1, ω(u) := −1
      else
        ω(u) := ω(v) := 0, t := u, Kontrola-DELETE(t)
      endif
    else
      u1 := levy(u), Dvojita-rotace(v, u, u1)
      if ω(u1) = 1 then
        ω(u) := 0, ω(v) := −1
      else
        if ω(u1) := 0 then
          ω(u) := 0, ω(v) := 0
        else
          ω(u) := 1, ω(v) := 0
```

```
          endif
        endif
        ω(u1) := 0, t := u1, Kontrola-Delete(t)
      endif
    else
      if ω(v) = 0 then
        ω(v) := 1
      else
        ω(v) := 0, t := v, Kontrola-DELETE(t)
      endif
    endif
else
    if ω(v) = −1 then
      if ω(u) ≤ 0 then
        Rotace(v, u)
        if ω(u) = 0 then
          ω(v) := −1, ω(u) := 1
        else
          ω(u) := ω(v) := 0, t := u, Kontrola-DELETE(t)
        endif
      else
        u2 := pravy(u), Dvojita-rotace(v, u, u2)
        if ω(u2) = 1 then
          ω(u) := −1, ω(v) := 0
        else
          if ω(u2) := 0 then
            ω(u) := 0, ω(v) := 0
          else
            ω(u) := 0, ω(v) := 1
          endif
        endif
        ω(u2) := 0, t := u2, Kontrola-Delete(t)
      endif
    else
      if ω(v) = 0 then
        ω(v) := −1
      else
        ω(v) := 0, t := v, Kontrola-DELETE(t)
      endif
    endif
endif
```

V operaci **DELETE** se může stát, že procedury **Rotace** nebo **Dvojita-rotace** jsou volány až $\log(|S|)$-krát. To je výrazný rozdíl proti operaci **INSERT**. Proto operace **DELETE** je pomalejší než operace **INSERT**, i když asymptoticky jsou stejně rychlé. Korektnost se ověří přímo.

Věta. *Datová struktura AVL-strom umožňuje implementaci operací* **MEMBER**, **INSERT** *a* **DELETE**, *které vyžadují čas $O(\log(|S|))$ (kde S je reprezentovaná množina). Operace* **INSERT** *zavolá nejvýše jednu proceduru* **Rotace** *nebo* **Dvojita-rotace**.

Červeno-černé stromy

Binární vyhledávací strom T reprezentující množinu S, jehož vrcholy jsou obarveny červeně nebo černě (každý vrchol má právě jednu barvu) tak, že jsou splněny podmínky:

listy jsou obarveny černě,
když v je vrchol obarvený červeně, pak je buď kořen stromu nebo jeho otec je obarven černě,
všechny cesty z kořene do listů mají stejný počet černých vrcholů

se nazývá červeno-černý strom.

Nejprve ukážeme, že červeno-černé stromy jsou vyvážené stromy, tj.

$$\text{hloubka}\,(T) = O\,(\log\,(|S|))\,.$$

Předpokládejme, že T je červeno-černý strom, který má na cestě z kořene do listu právě k černých vrcholů. Pak pro počet vrcholů $\#T$ stromu T platí

$$2^k - 1 \leq \#T \leq 2^{2k} - 1.$$

Nejmenší takový strom má všechny vrcholy černě obarvené a je to úplný pravidelný binární strom o výšce $k-1$, což dává dolní odhad. Největší takový strom má všechny vrcholy v sudých hladinách obarveny červeně a v lichých hladinách černě, je to úplný pravidelný binární strom o výšce $2k-1$ a tím je dán horní odhad. Tedy $k \leq \log\,(1 + \#T) \leq 2k$. Protože velikost S je počet vnitřních vrcholů, dostáváme, že $\#T = 2|S| + 1$. Z vlastností červeno-černých stromů plyne, že

$$k \leq \text{hloubka}\,(T) \leq 2k.$$

Tedy

Tvrzení. *Když červeno-černý strom T reprezentuje množinu S, pak*

$$\text{hloubka}\,(T) \leq 2\log\,(2|S| + 2) = 2\,(1 + \log\,(|S| + 1))\,.$$

Pro červeno-černé stromy navrhneme algoritmy realizující operace z uspořádaného slovníkového problému. Operace **MEMBER** pro červeno-černé stromy je stejná jako pro nevyvážené binární vyhledávací stromy. Operace **INSERT** a **DELETE** mají dvě části: nejprve se provede operace **INSERT** nebo **DELETE** pro nevyvážené binární vyhledávací stromy a pak následují vyvažovací operace, které zajistí, že výsledný strom splňuje podmínky pro červeno-černé stromy (stejné schéma jako pro AVL-stromy). Schéma operací **JOIN** a **SPLIT** bude vycházet z jejich realizací v (a, b)-stromech. V operaci **JOIN** prohledáváním nalezneme místo, kde se stromy dají spojit (a aplikujeme operaci **JOIN** pro nevyvážené binární vyhledávací stromy), a pak použijeme vyvažovací operace. Algoritmus operace **SPLIT** rozdělí červeno-černý strom do několika menších podle cesty vyhledávající x (podobně jako v (a, b)-stromech) a na tyto stromy pak aplikuje operaci **JOIN** a zkonstruuje hledané červeno-černé stromy. Algoritmy pro operace **MIN** a **MAX** jsou stejné jako pro nevyvážené binární vyhledávací stromy.

Nejprve popíšeme vyvažovací operace. Dvojice (T, v) se nazývá 2-parciální červeno-černý strom, když T je binární vyhledávací strom, každý vrchol je obarven červeně nebo černě, v je vnitřní vrchol stromu T obarvený červeně a platí:

listy jsou obarveny černě,
když t je vrchol obarvený červeně, pak je buď kořen stromu nebo $t = v$ nebo jeho otec je obarven černě,
všechny cesty z kořene do listů mají stejný počet černých vrcholů.

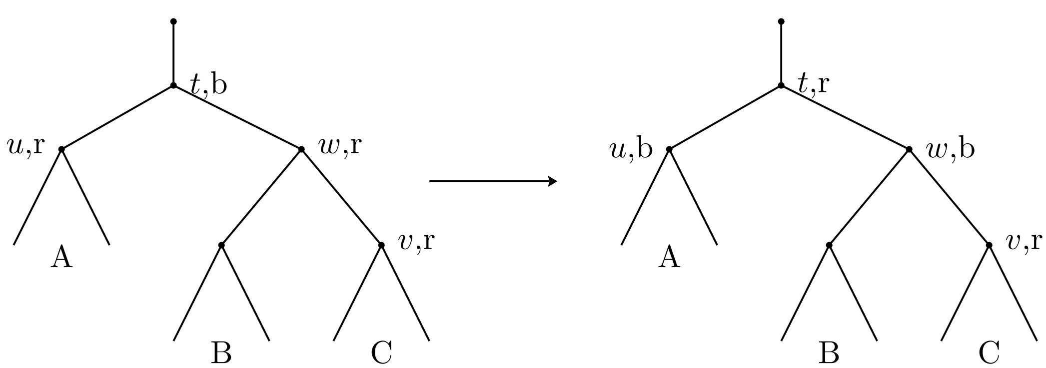
Vyvažování 2-parciálního červeno-černého stromu (T', v) provádí procedura **Vyvaz-INSERT**(v). Po jejím provedení buď dostaneme červeno-černý strom nebo je procedura **Vyvaz-INSERT** zavolána na vrchol v' takový, že (T', v') je 2-parciální červeno-černý strom a v' je děd v (tj. je o dvě hladiny blíž ke kořeni než vrchol v).

Obarvení je realizováno rozšířením struktury vrcholu v o boolskou proměnnou $b\,(v)$, kde $b\,(v) = 0$ znamená, že v je obarven červeně, a $b\,(v) = 1$ znamená, že v je obarven černě.

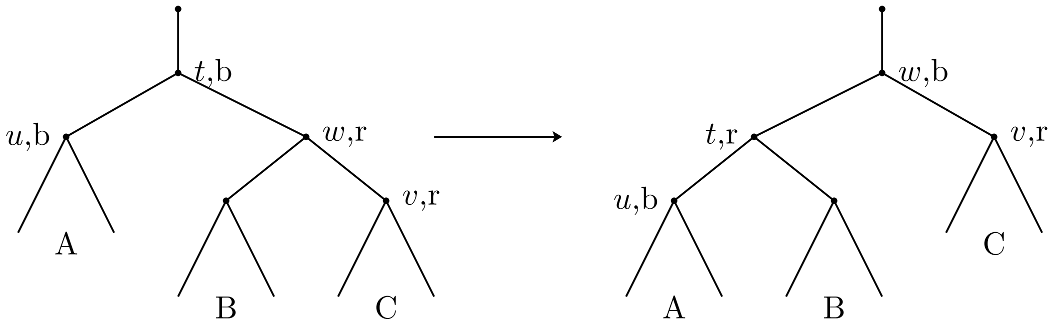
Popíšeme proceduru **Vyvaz-INSERT**(v) (předpokládáme, že v je obarven červeně). Pro zjednodušení $s(v) = \mathit{levy}$, když $v =$ levy(otec(v)), a $s(v) = \mathit{pravy}$ pro $v =$ pravy (otec (v)).

```
  Vyvaz-INSERT(v).
if v není kořen T' a b (otec (v)) = 0 then
  if otec (v) je kořen then
    b (otec (v)) := 1
  else
    w := otec (v), u := bratr (w)
    if b (u) = 0 then
      t := otec (w), b (w) := 1, b (u) := 1
      b (t) := 0, Vyvaz-INSERT(t) (Viz Obr. 1)
    else
      t := otec (w)
      if s (w) = s (v) then
        Rotace(t, w), b (t) := 0, b (w) := 1 (Viz Obr. 2)
      else
        Dvojita-rotace(t, w, v), b (t) := 0, b (v) := 1 (Viz Obr. 3)
      endif
    endif
  endif
endif
```

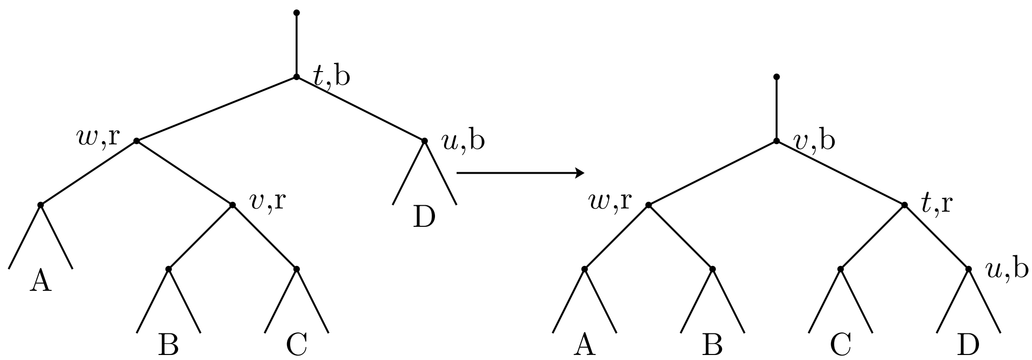
Na obrázku b značí černou barvu a r značí červenou barvu. Otec vrcholu w je označen t.



Obr. 1



Obr. 2



Obr. 3

2-parciální červeno-černé stromy vznikají při operacích **INSERT** a **JOIN.** Při operaci **DELETE** se poruší struktura červeno-černých stromů jiným způsobem a vznikne 3-parciální červeno-černý strom.

Řekneme, že dvojice (T, v) je 3-parciální červeno-černý strom, když T je binární vyhledávací strom, každému vrcholu je přiřazena právě jedna z dvojice barev červená – černá, v je vrchol ve stromu T a platí následující podmínky:

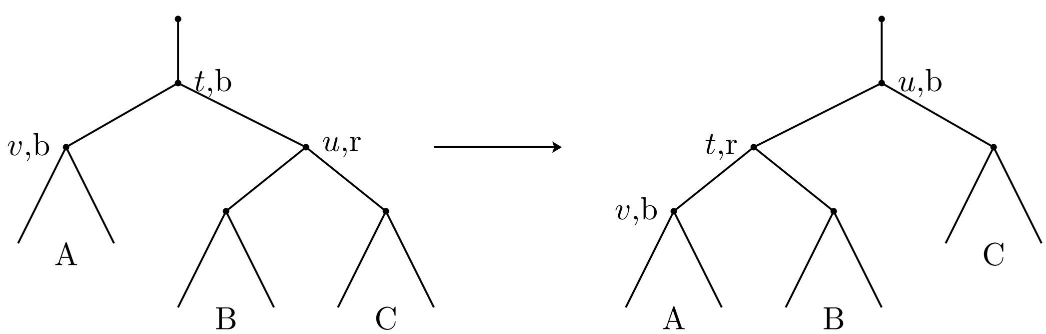
listy a vrchol v jsou obarveny černě,
když t je vrchol obarvený červeně, pak je buď kořen stromu nebo jeho otec je obarven černě,
existuje číslo k takové, že všechny cesty z kořene do listů, které neobsahují vrchol v, obsahují právě k černých vrcholů, a všechny cesty z kořene do listů procházející vrcholem v obsahují $k-1$ černých vrcholů.

Popíšeme proceduru **Vyvaz-DELETE**(v), která se použije na 3-parciální červeno-černý strom (T, v), když v není jeho kořen. Výsledkem procedury bude buď červeno-černý strom nebo zavolání procedury **Vyvaz-DELETE**(v'), kde v' je otcem vrcholu v. Z faktu, že když (T, v) je 3-parciální červeno-černý strom a v je jeho kořen, pak T je červeno-černý strom, plyne, že aplikací **Vyvaz-DELETE**(v) na 3-parciální červeno-černý strom (T, v) dostaneme pomocípřípadně několika volání procedury **Vyvaz-DELETE** červeno-černý strom.

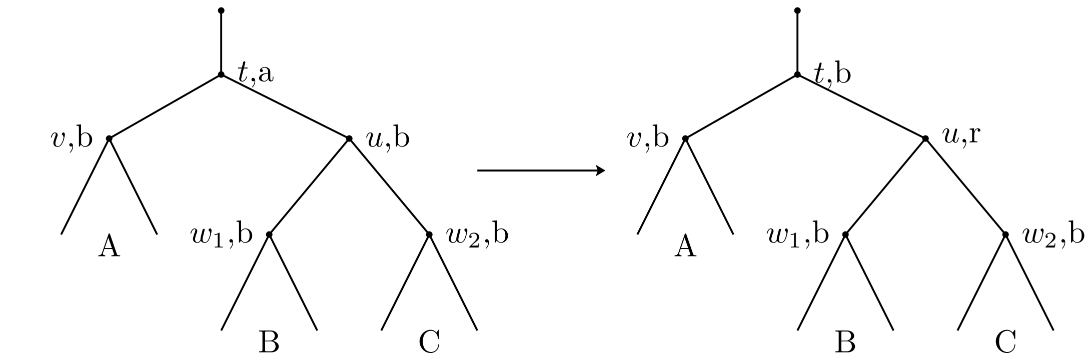
Vyvaz-DELETE(v)

$u :=$ bratr (v), $t :=$ otec (v)
if $b\,(u) = 0$ **then**
  Rotace(t, u), $b\,(u) := 1$, $b\,(t) := 0$, $u :=$ bratr (v)
endif
(Viz Obr. 4, Komentář: nyní $b\,(u) = 1$ a $b\,(t) = 0$)
w_1 je syn u takový, že $s\,(v) = s\,(w_1)$, $w_2 :=$ bratr (w_1)
if $b\,(w_1) = b\,(w_2) = 1$ **then**
  $b\,(u) := 0$
  if $b\,(t) := 0$ **then**
    $b\,(t) := 1$
  else
    if t není kořen stromu **then**
      $v := t$, **Vyvaz-DELETE**(v)
    endif
  endif (Viz Obr. 5)
else
  if $b\,(w_1) = 1$ **then**
(Komentář: $b\,(w_2) = 0$)
    Rotace(t, u), $b\,(w_2) := 1$, $b\,(u) := b\,(t)$, $b\,(t) := 1$ (Viz Obr. 6)
  else
    Dvojita-rotace(t, u, w_1), $b\,(w_1) := b\,(t)$, $b\,(t) := 1$ (Viz Obr. 7)
  endif
endif

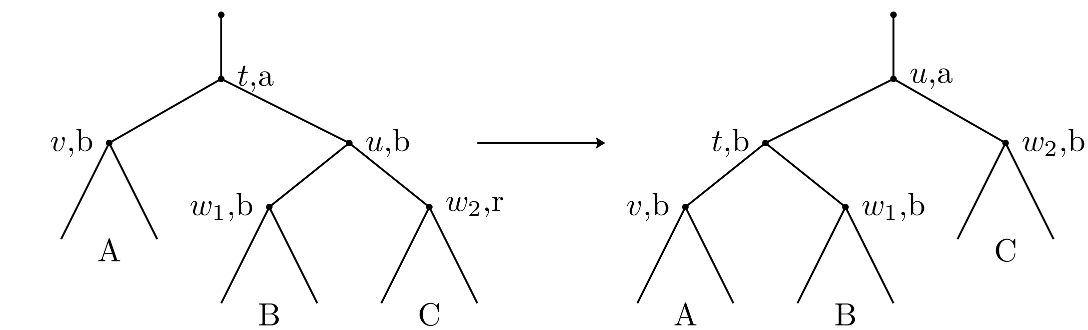
V následujících obrázcích jsou vrcholy, které nemají specifikovanou barvu (mohou být jak červené tak černé). Tyto barvy budeme označovat a, a'. Důvod je, že se tato barva může přenést do cílového stromu, ale i na jiný vrchol. V tomto smyslu jsou tyto barvy určeny vstupním stromem a specifikují tyto barvy v cílovém stromě. V Obr. 5 se barva a v cílovém stromě neobjevuje.



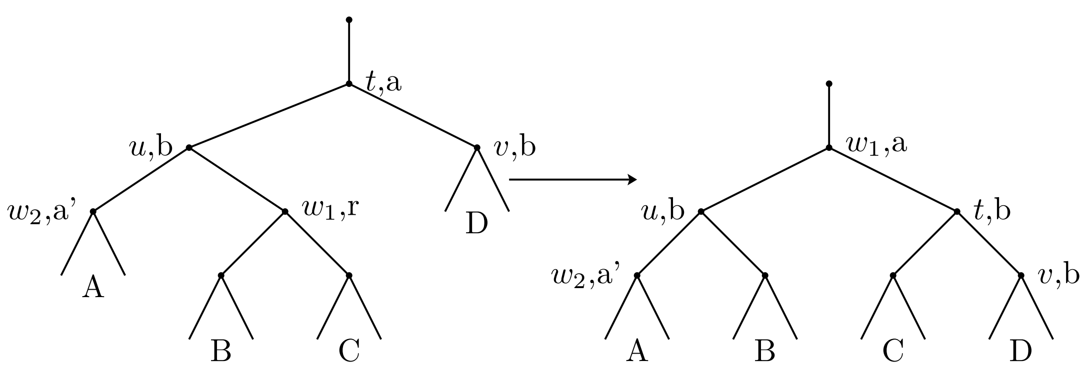
Obr. 4



Obr. 5



Obr. 6



Obr. 7

Nyní popíšeme algoritmy realizující operace **INSERT**, **DELETE**, **JOIN3** a **SPLIT** pro červeno-černé stromy. Předpokládejme, že T je červeno-černý strom reprezentující množinu S a provádíme operaci **INSERT**(x) pro $x \notin S$. Když operace **INSERT**(x) pro nevyvážené binární vyhledávací stromy vytvoří strom T', kde vrchol v reprezentuje x, pak v obarvíme červeně a syny v (jsou to listy) obarvíme černě. Dostáváme, že (T', v) je 2-parciální červeno-černý strom, a pak aplikujeme proceduru **Vyvaz-INSERT**.

Operace **INSERT** v červeno-černých stromech volá nejvýše $2 + \log(|S|)$-krát proceduru **Vyvaz-INSERT** a provede nejvýše jednu rotaci nebo dvojitou rotaci.

Operace **DELETE** je řešena stejným způsobem jako operace **INSERT**, ale při operaci **DELETE** je porušena třetí podmínka v definici červeno-černých stromů a vyvažování je technicky náročnější. Předpokádejme, že T je červeno-černý strom. Když chceme provést operaci **DELETE**, pak nejprve provedeme algoritmus **DELETE** pro nevyvážené binární vyhledávací stromy. Při provádění jsme odstranili vrchol u a jeho syna w, který je list. Na místo vrcholu u se dostal jeho druhý syn v, který obarvíme černě. Pak jsou splněny první dvě podmínky v definici červeno-černých stromů a pokud vrchol u nebo vrchol v byl obarven červeně, pak je splněna i třetí podmínka. Pokud vrchol u i vrchol v byly obarveny černě, pak každá cesta z kořene do listu obsahující vrchol v má o jeden černý vrchol méně než cesta z kořene do listu neobsahující vrchol v (chybí černý vrchol u), a tedy (T, v) je 3-parciální červeno-černý strom. Nyní aplikujeme proceduru **Vyvaz-DELETE**. Analýza poskytuje rychlý test na to, zda vznikne červeno-černý strom nebo 3-parciální červeno-černý strom (pak v je list).

Mějme červeno-černé stromy T_1 a T_2 reprezentující množiny S_1 a S_2 a mějme prvek $x \in U$ takový, že $\max S_1 < x < \min S_2$. Nejprve zajistíme, že kořeny T_1 i T_2 jsou obarveny černě. Předpokládejme, že k_i je počet černých vrcholů na cestě z kořene do listů ve stromě T_i pro $i = 1, 2$. Když $k_1 = k_2$, pak stačí provést **JOIN3**(T_1, x, T_2) pro nevyvážené binární vyhledávací stromy (kořen obarvíme červeně). Problém je, když $k_1 \neq k_2$. Například předpokládejme, že $k_1 > k_2$. Pak začneme v kořeni stromu T_1 a jdeme po pravých synech dolů tak dlouho, až nalezneme černý vrchol v takový, že všechny cesty z v do listů v T_1 obsahují právě k_2 černých vrcholů. Pak provedeme **JOIN3** pro nevyvážené binární vyhledavací stromy na podstrom T_1 určený vrcholem v, na x a na T_2. Kořen w vzniklého stromu obarvíme červeně a tento strom vložíme do T_1 místo podstromu určeného vrcholem v. Pak (T_1, w) je 2-parciální červeno-černý strom a aplikujeme proceduru **Vyvaz-INSERT**. Případ $k_2 > k_1$ se řeší symetricky.

Algoritmus pro operaci **SPLIT** je velmi podobný algoritmu pro (a, b)-stromy. Vyhledáváme vrchol reprezentující x. Když jsme ve vrcholu t a pokračujme akcí $t :=$ levy(t), pak dvojici key(t) a podstrom T určený pravým synem t vložíme do zásobníku Z_2, když pokračujeme akcí $t :=$ pravy(t), pak do zásobníku Z_1 vložíme dvojici podstrom T určený levým synem T a key(t). Když key$(t) = x$, pak do Z_1 vložíme podstrom určený levým synem t a do Z_2 podstrom určený pravým synem t. Když t je list, pak do Z_1 i Z_2 vložíme jednoprvkové stromy. Ze zásobníku Z_1 pomocí operace **JOIN3** vytvoříme strom T_1 a ze zásobníku Z_2 pomocí operace **JOIN3** dostaneme strom T_2.

Nyní popíšeme algoritmy pro tyto operace.

  INSERT(x)
Vyhledej(x)
if t je list **then**
  t se změní na vnitřní vrchol, key$(t) := x$
  pro vrchol t vytvořme syny levy(t) a pravy(t)
  $b(t) := 0$, $b(\text{levy}(t)) := 1$, $b(\text{pravy}(t)) := 1$, **Vyvaz-INSERT**(t)
endif

  DELETE(x)
Vyhledej(x)
if t není list **then**
  $vyv := false$
  if levy(t) je list **then**

```
        v := pravy (t)
        if b (t) = 1 a b (v) = 1 then
            vyv := true
        endif
        odstraníme vrchol levy (t), otec (v) := otec (t)
        if t = levy (otec (t)) then
            levy (otec (t)) := v
        else
            pravy (otec (t)) := v
        endif
        b (v) := 1, odstraníme vrchol t
    else
        u := levy (t)
        while pravy (u) není list do u := pravy (u) enddo
        key (t) := key (u), v := levy (u)
        if b (u) = 1 a b (v) = 1 then
            vyv := true
        endif
        odstraníme vrchol pravy (u), otec (v) := otec (u)
        if u = levy (otec (u)) then
            levy (otec (u)) := v
        else
            pravy (otec (u)) := v
        endif
        b (v) := 1, odstraníme vrchol u
    endif
    if vyv then Vyvaz-DELETE(v) endif
endif
```

JOIN3(T_1, x, T_2)

```
if b (kořen T1) = 0 then b (kořen T1) := 1 endif
if b (kořen T2) = 0 then b (kořen T2) := 1 endif
k1 je počet černých vrcholů v T1 z kořene do listů
k2 je počet černých vrcholů v T2 z kořene do listů
if k1 ≥ k2 then
    t := kořen T1, i := k1 − k2
    while i > 0 do
        t := pravy (t)
        if b (t) = 1 then i := i − 1 endif
    enddo
    vytvoř vrchol u, b (u) := 0, key (u) := x
    if t není kořen T1 then
        otec (u) := otec (t), pravy (otec (t)) := u
    endif
    otec (t) := u, otec (kořen T2) := u
    pravy (u) := kořen T2, levy (u) := t, Vyvaz-INSERT(T1, u)
else
    t := kořen T2, i := k2 − k1
    while i > 0 do
```

```
      t := levy (t)
      if b (t) = 1 then i := i − 1 endif
   enddo
   vytvoř vrchol u, b (u) := 0, key (u) := x
   otec (u) := otec (t), levy (otec (t)) := u, otec (t) := u
   otec (kořen T1) := u, levy (u) := kořen T1
   pravy (u) := t, Vyvaz-INSERT(T2, u)
endif
```

SPLIT(x)

```
Z1 a Z2 jsou prázdné zásobníky, t := kořen T
while key (t) ≠ x a t není list do
   if key (t) > x then
      vlož (key (t), pravy (t)) do Z2, t := levy (t)
   else
      vlož (levy (t), key (t)) do Z1, t := pravy (t)
   endif
enddo
if key (t) = x then
   Výstup: x ∈ S, T1 je podstrom T určený levy (t)
   T2 je podstrom T určený pravy (t)
else
   Výstup: x ∉ S, T1 a T2 jsou jednoprvkové stromy
endif
while Z1 ≠ ∅ do
   (t, x) je na vrcholu Z1, odstraň (t, x) ze Z1
   T' je podstrom T určený t, T1 :=JOIN3(T', x, T1)
enddo
while Z2 ≠ ∅ do
   (x, t) je na vrcholu Z2, odstraň (x, t) ze Z1
   T' je podstrom T určený t, T2 :=JOIN3(T2, x, T')
enddo
```

Korektnost algoritmů je vidět z obrázků. Všimněme si při operaci **DELETE**, že když u je obarven červeně, pak po provedení **Rotace**(t, u) bude (T, v) opět 3-parciální červeno-černý strom a vrchol t bude obarven červeně. Pak z Obr. 5 je vidět, že dostaneme červeno-černý strom. Tedy můžeme shrnout:

Věta. *Algoritmy operací* **MEMBER**, **INSERT**, **DELETE**, **MIN**, **MAX**, **JOIN3** *a* **SPLIT** *pro červeno-černé stromy vyžadují v nejhorším případě čas* $O(\log(|S|))$, *kde* S *je reprezentovaná množina. Operace* **INSERT** *a* **JOIN3** *zavolají nejvýše jednou buď* **Rotace** *nebo* **Dvojita-rotace** *a operace* **DELETE** *zavolá nejvýše dvakrát* **Rotace** *nebo* **Rotace** *a* **Dvojita-rotace**.

Všimněte si, že operace **JOIN3** ve skutečnosti vyžaduje čas $O(|k_1 - k_2| + 1)$. Protože Z_1 a Z_2 obsahují nejvýše $\log(|S|)$ položek, tak se odhad časové složitosti operace **SPLIT** provede stejným způsobem jako v (a, b)-stromech. V ostatních případech je odhad časové složitosti vidět z toho, že hloubka $(T) = O(\log(|S|))$ a akce na každé hladině vyžadují jen $O(1)$ času.

Pokud chceme mít i algoritmus pro operaci **ord**(k), pak musíme rozšířit strukturu o funkci p. Pak lze použít přímo algoritmus pro **ord**(k) v nevyvážených binárních vyhledávacích stromech. Připomeňme si, že procedury **Rotace** a **Dvojita-rotace** mohou aktualizovat funkci p v čase $O(1)$. Proto dostáváme

Věta. *Algoritmy operací* **MEMBER**, **INSERT**, **DELETE**, **MIN**, **MAX**, **JOIN3**, **SPLIT** *a* **ord**(k) *pro rozšířenou strukturu červeno-černých stromů vyžaduje v nejhorším případě čas* $O(\log(|S|))$, *kde* S *je reprezentovaná množina. Operace* **INSERT** *a* **JOIN3** *zavolají nejvýše jednou buď* **Rotace** *nebo* **Dvojita-rotace** *a operace* **DELETE** *zavolá nejvýše dvakrát* **Rotace** *nebo jednou* **Rotace** *a* **Dvojita-rotace**.

Vzniká otázka, proč se tolik pozornosti věnuje procedurám **Rotace** a **Dvojita-rotace**. Sice vyžadují čas $O(1)$, ale jsou to nejsložitější akce vyžadující nejvíce času. V mnoha aplikacích (používají se hlavně ve výpočetní geometrii), tvar stromu spolu s parametry nesou ještě další zakódované informace. Při změně tvaru stromu je třeba je přepočítat. **Rotace** a **Dvojita-rotace** mění tvar stromu, kdežto posun směrem ke kořeni pouze mění obarvení. V tomto případě pak **Rotace** nebo **Dvojita-rotace** vyžaduje čas $O(|S|)$ (obvykle je třeba prohlédnout celý strom) a nikoliv $O(1)$.

Váhově vyvážené stromy

V osmdesátých letech se ve výpočetní geometrii hodně používaly $BB(\alpha)$-stromy, proto se o nich alespoň orientačně zmíníme. Mějme reálné číslo α takové, že $\frac{1}{4} < \alpha \leq \frac{\sqrt{2}}{2}$. Pro strom T označme $p(T)$ počet listů ve stromu T. Binární vyhledávací strom T reprezentující množinu S se nazývá $BB(\alpha)$-strom, když pro každý vnitřní vrchol v platí:

$$\alpha \leq \frac{p(T_l)}{p(T_v)} = 1 - \frac{p(T_r)}{p(T_v)} \leq 1 - \alpha$$

kde T_v je podstrom T určený vrcholem v, T_l je podstrom T určený levým synem vrcholu v, T_r je podstrom T určený pravým synem vrcholu v. Platí

Tvrzení. *Když T je $BB(\alpha)$-strom reprezentující n-prvkovou množinu, pak*

$$\text{hloubka}(T) \leq 1 + \frac{\log(n+1) - 1}{\log \frac{1}{1-\alpha}}.$$

Důsledek je, že $BB(\alpha)$-stromy patří do skupiny vyvážených binárních vyhledávacích stromů. Vyvažování se provadí opět pomocí **Rotace** a **Dvojita-rotace** a popisuje ho následující technické tvrzení.

Tvrzení. *Pro každé α existuje konstanta d taková, že $\alpha < d < 1 - \alpha$ a pro každý binární vyhledávací strom T s kořenem t splňující podmínky*

(1) *podstromy T_l a T_r stromu T určené levým a pravým synem t jsou $BB(\alpha)$-stromy;*
(2) $\frac{p(T_l)}{p(T)} < \alpha$, *ale* $\alpha \leq \frac{p(T_l)}{p(T)-1} \leq 1 - \alpha$ *nebo* $\alpha \leq \frac{p(T_l)+1}{p(T)+1} \leq 1 - \alpha$

platí:
když $\rho \leq d$ a provedeme **Rotace**$(t, \text{pravy}(t))$, *nebo když $\rho > d$ a provedeme proceduru* **Dvojita-rotace**$(t, \text{pravy}(t), \text{levy}(\text{pravy}(t)))$, *pak dostaneme $BB(\alpha)$-strom (zde $\rho = \frac{p(T')}{p(T_r)}$ a T' je určen levým synem pravého syna kořene t).*

Toto tvrzení a jeho symetrické verze jednoznačně ukazují, jak vyvažovat $BB(\alpha)$-stromy při aktualizačních operacích (podstrom $BB(\alpha)$-stromu je $BB(\alpha)$-strom). Pak dostáváme:

Věta. *Implementace operací* **MEMBER**, **INSERT** *a* **DELETE** *v* $BB(\alpha)$*-stromech vyžaduje v nejhorším případě čas* $O(\log(|S|))$, *kde* S *je reprezentovaná množina.*

Obliba $BB(\alpha)$-stromů byla zapříčiněna platností následující věty, která je analogií věty o vyvažovacích operacích pro (a, b)-stromy.

Věta. *Když* α *je reálné číslo takové, že* $\frac{1}{4} < \alpha < 1 - \frac{\sqrt{2}}{2}$, *pak existuje konstanta* $c > 0$ *závislá jen na* α *taková, že každá posloupnost operací* **INSERT** *a* **DELETE** *o délce* m *aplikovaná na prázdný* $BB(\alpha)$*-strom volá nejvýše* cm *procedur* **Rotace** *a* **Dvojita-rotace**.

Historický přehled:

(a, b)-stromy zavedli Bayer a McGreght (1972),
věty o počtu vyvažovacích operací pro (a, b)-stromy dokázali Huddleston a Mehlhorn (1982).
A-sort analyzovali Guibas, McGreight, Plass a Roberts (1977).

Analýza interpolačního vyhledávání pochází od Perla, Itai a Avniho (1978),
kvadratické vyhledávání analyzovali Perl a Reingold (1977).

Adelson-Velskij a Landis (1962) definovali AVL-stromy,
červeno-černé stromy definovali Guibas a Sedgewick (1978),
verze algoritmu **DELETE** pochází od Tarjana (1983). $BB(\alpha)$-stromy zavedli Nievergelt a Reingold (1973),
věty o jejich vyvažování dokázali Blum a Mehlhorn (1980).
Řada uživatelů používá AVL-stromy, což je důsledkem jejich dobré efektivity a faktu, že po dlouhou dobu to byla jediná efektivní datová struktura. Zdá se však, že červeno-černé stromy jsou efektivnější a v současné době jsou nejpoužívanějším typem binárních vyhledávacích stromů.